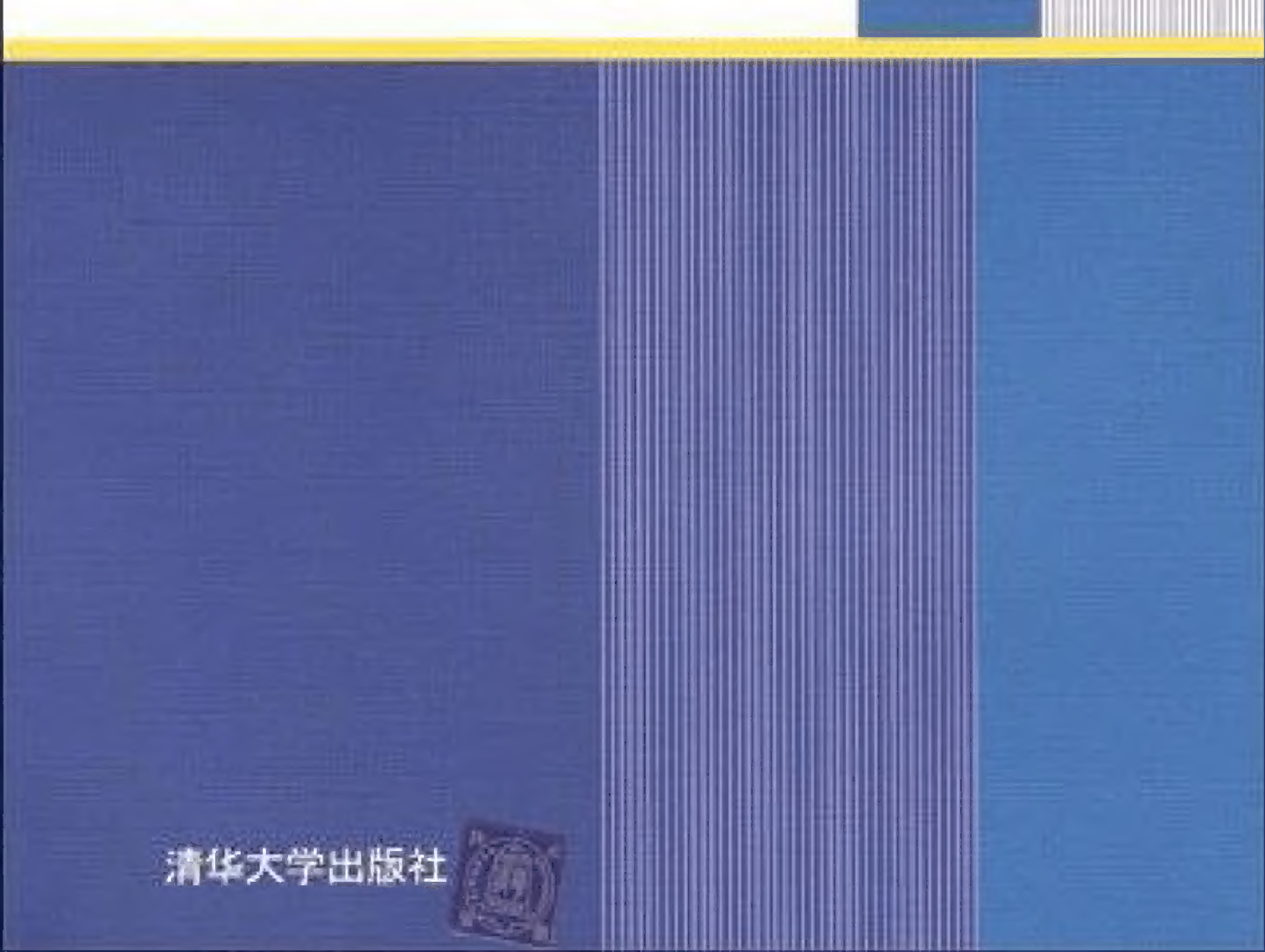
郁莲 编著

软件测试方法与实践



软件测试方法与实践

郁 莲 编著

清华大学出版社
北 京

内 容 简 介

本书系统介绍现代软件测试的基本原理与一般方法。全书共分10章，内容包括软件测试概述、白盒测试、黑盒测试、软件测试覆盖分析、单元测试与集成测试、JUnit测试工具、回归测试、基于状态的软件测试技术、面向对象的应用测试、Web应用软件测试技术。各章均有总结、思考与练习题、课后作业和进一步阅读材料，以便巩固加深所学的知识。

本书可作为计算机科学软件工程专业的本科高年级学生及研究生的教科书，以及从事软件测试工作的技术人员的参考书。

图书在版编目(CIP)数据

软件测试方法与实践/郁莲编著. —北京：清华大学出版社，2008.11
ISBN 978-7-302-18458-4

Ⅰ. 软… Ⅱ. 郁… Ⅲ. 软件—测试 Ⅳ. TP311.5

中国版本图书馆CIP数据核字(2008)第132659号

责任编辑：丁 岭 李 晔
责任校对：梁 毅
责任印制：

出版发行：清华大学出版社　　**地　址**：北京清华大学学研大厦A座
http://www.tup.com.cn　　**邮　编**：100084
社 总 机：010-62770175　　**邮　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　销：全国新华书店
开　本：185×230　　**印　张**：14.75　　**字　数**：326千字
版　次：2008年11月第1版　　**印　次**：2008年11月第1次印刷
印　数：1～ 000
定　价： .00元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。
联系电话：010-62770177 转 3103　　产品编号：

前言

在高度信息化的今天，信息技术已经成为社会发展的第一生产力，软件则是信息技术中最重要的组成部分。近年来，软件产业在很多国家都成为了国民经济的主导产业。但随着软件的规模和复杂性的大幅度提升，软件不可靠性的矛盾也变得日益突出，因此如何保证软件的质量成为了必须解决的问题。在20世纪，由于需求和认识等方面的原因，更多的人只是关注软件开发，而软件测试一直没有得到足够的重视，发展比较缓慢。随着软件质量保证理论与技术的快速发展，软件测试逐渐受到越来越广泛的重视，并正在形成一种产业，从业人员的数量也在大幅度增加。目前中国有一千多家软件评测中心，从事软件测试的人员有数万人，但仍然有约二十万的人才空缺。这些紧缺人才并不是只会点点鼠标的测试操作者，而是具有与开发人员相同甚至更高能力的测试设计师和分析员。

1. 预备知识要求和目标

本书既可作为初次接触软件测试的读者系统学习的入门教材，也可作为具有一定经验的测试人员随时翻阅的工具书。本书难度适中，希望读者通过阅读和学习，能够了解软件测试的重要性，掌握基本的软件测试技术。不论是哪类读者，要深入理解本书的内容，软件工程的基础知识都是必需的。

另外，最后两章涉及面向对象的应用测试和Web应用软件测试，如果读者具有一定的面向对象开发基础和Web应用开发基础，便能够更加透彻地理解这两章的内容。当然，这并不是必需的，即使没有这方面的经验，读者也可以利用章节最后列出的进一步阅读材料了解相关的知识。

2. 本书章节安排

软件测试的基础包括黑盒测试技术和白盒测试技术，除此之外，随着人们对软件及软件错误认识的不断深入，新的测试方法也应运而生。本书在重点讲解软件测试基本技术的同时，也对一些新的软件测试技术进行了介绍。

第1章　软件测试概述。软件测试是为了确认软件做了所期望的事情(Do the Right Thing)，另一方面是确认软件以正确的方式来做了这个事件(Do it Right)。软件测试不仅



是在测试软件产品本身，而且还包括软件开发的过程。这一章将介绍软件测试基本概念、软件测试目的、软件测试类型、软件测试原则、软件测试现状与挑战和测试人员职业发展与素质。

第 2 章　白盒测试。白盒测试(White-box Testing)是一种基于源程序或代码的测试方法，包括静态和动态两种类型。静态方法是指按一定步骤直接检查源代码以发现错误，也称为代码检查法；动态方法是指按一定步骤生成测试用例并驱动被测程序运行以发现错误。这一章介绍了基本路径测试、条件测试、数据流测试及循环测试等动态方法，以及有桌面检查、代码审查及走查等静态方法。

第 3 章　黑盒测试。黑盒测试(Black－box Testing)注重于测试软件的功能性需求，即黑盒测试需要软件工程师生成输入条件集来检测程序所有的功能需求。黑盒测试用于配合白盒测试发现其他类型的错误，包括功能错误或遗漏、界面错误、数据结构或外部数据库访问错误、性能错误和初始化和终止错误。这一章对黑盒测试技术进行了详细的讲解。

第 4 章　软件测试覆盖分析。在掌握了白盒测试与黑盒测试技术后，还需要一种方式来了解测试已经执行的程度，即本章要介绍的软件测试覆盖分析(Coverage Analysis)技术。在测试计划阶段，测试者确定用何种测试覆盖分析及相应的覆盖率；在测试执行阶段，将根据既定的覆盖率来检查是否进行了足够的测试。这一章将主要地介绍面向白盒测试技术的代码覆盖分析(控制流覆盖和数据流覆盖)，并简要地介绍几种面向黑盒测试技术的覆盖分析。

第 5 章　单元测试与集成测试。分阶段测试是一种基本的测试策略，其中单元测试(Unit Testing)是对最小的软件设计单元(模块或源程序单元)的验证工作，集成测试(Integration Testing)把单独的软件模块结合在一起，作为一个群接受测试。这一章将介绍单元测试与集成测试概念、目标及过程。

第 6 章　JUnit 测试工具。JUnit 是一个开源的 Java 编程语言的单元测试框架，最初由 Erich Gamma 和 Kent Beck 编写。它在代码驱动单元测试框架家族里无疑是最为成功的一例。学习 JUnit 不但可以掌握一种有力的测试工具，更能帮助读者深入了解测试过程，对今后使用其他测试工具甚至开发测试工具都有极大的帮助。这一章介绍了 JUnit 的安装与使用，并讲解了 JUnit 自身的设计。

第 7 章　回归测试。在软件开发、维护、升级的不断演进过程中，由于各种各样的原因例如功能性/非功能性需求的改变、技术更新和升级了的软硬件平台，使得软件系统经常发生改变。这使得回归测试变得非常重要。回归测试(Regression Testing)是对之前已测试过、经过修改了的程序进行的重新测试，以保证该修改没有引入新的错误或者由于更改而发现之前未发现的错误，这是这一章的主要内容。

第 8 章　基于状态的软件测试技术。基于状态的软件测试是一种基于模型的测试技术。通过建立描述系统行为的状态机来自动生成测试用例。这一章介绍基于状态模型的软件测试技术，这在对实时系统、嵌入式系统、Web 交互性系统中具有广泛的应用。

第 9 章　面向对象的应用测试。面向对象是现代软件开发的主流，面向对象的应用测试

是相关活动的集合，是为了发现存在于OOA、OOD、类、方法(操作)以及类间交互方面的错误。为了完成这些活动，在面向对象的应用中要使用包括静态评审和动态执行测试在内的测试策略。这一章详细论述了面向对象的软件测试技术，以及它与传统的软件测试方法的区别。

第10章　Web应用软件测试技术。Web应用测试是测试每个Web应用质量的纬度(Dimension)，目的是发现错误或者发现导致质量失败问题。测试集中在内容、功能、结构、易用性、导航、性能、兼容性、互操作、容量、安全等方面。测试应该和Web应用设计的评审相结合。这一章对Web应用测试进行了详细的讲解。

3. 思考练习与进一步阅读

不管是课堂讲授还是自学，简单的练习对加强理解都是特别有用的，因此本书的每一章后面都有一组思考与练习。这些题目难度适中，既可以在课堂内完成，以帮助老师了解学生对这些内容的理解掌握程度，也可以做为课后的练习，帮助学生复习和巩固学到的知识。

对于那些不满足于课本内容的学生和读者，一系列相关的扩展阅读无疑像是科学探索中一盏盏引路的明灯，因此每一章最后都列出了与本章内容有关的一些学术论文或书籍，供学有余力的读者进一步深入学习。

4. 错误

无论作者有多少发现错误的技巧，总会有一些错误没被发现。如果读者发现任何可能是错误的地方，欢迎提出纠正建议，通过电子邮件 lianyu@ss. pku. edu. cn 反馈给作者，以便在本书下一次印刷时更正。同时欢迎读者为下一个版本的补充练习或要求提出宝贵建议。衷心感谢读者的帮助。

5. 感谢

作者近几年来一直在北京大学软件与微电子学院从事软件测试技术的研究与授课，并参与了很多软件质量保证方面的课题研究与工程项目，与国内外知名的研究单位、软件公司建立了广泛的学术联系。感谢学院对我的工作给予的极大支持与重视，感谢学院领导与老师们对我讲授的“软件测试技术”课程的大力支持。

硕士研究生周季、相慧如、伍晓东、张曲艳、吴芳、张乐、张辉辉、邸晓峰、许夏、宋胜攀等同学参与了本书的校对与整理工作，付出了很多努力，在此表示感谢。感谢北京大学软件与微电子学院0509、0609和0709届所有选修过“软件测试技术”课的同学，你们的反馈意见对我的教学工作与本书的编写都有很大的帮助。

郁　莲

2008年6月

目录

Contents

第1章

软件测试概述

软件测试是软件质量保证的重要手段。有研究数据显示，国外软件开发机构40%的工作量花在软件测试上，软件测试费用占软件开发总费用的30%～50%。对于一些要求高可靠、高安全的软件，测试费用可能相当于整个软件项目开发所有费用的3～5倍。由此可见，要成功开发出高质量的软件产品，除了从思想上重视软件测试工作，还必须掌握测试技术，有效地实施测试工作。

本章的内容包括软件测试基本概念、软件测试目的、软件测试类型、软件测试原则、软件测试现状与挑战以及测试人员职业发展与素质。

快速阅览：

什么是软件测试？ Myers (1979)定义测试(Testing)是执行程序的过程，其目的是发现错误。IEEE 610.12标准(1990)给出了两个测试定义：

(1) 在特定的条件下运行系统或构件，观察或记录结果，对系统的某个方面做出评价。

(2) 分析某个软件项以发现现存的和要求的条件之差别(即错误)并评价此软件项的特性。

由谁来负责软件测试？ 在测试初期，由软件工程师实施所有测试。然而，随着测试过程进行，测试专业人员应该加入进来。

为什么软件测试如此重要？ 没有经过测试的软件产品，无法知晓该软件产品运行时是否满足用户功能、性能需求，甚至导致最终用户生命、财产的损失。为了在把软件产品交付给用户之前尽可能多地发现错误(Error)，必须使用专业技术设计测试用例，进行系统化测试。

软件测试步骤各是什么？ 软件测试过程主要包括4个步骤：制定测试计划、生成测试用例、执行测试和分析测试结果。

有哪些工件形成？ 在一些情况下，会生成测试计划、测试用例和测试结果报告。测试结果存档以便将来软件维护时使用。

如何确保我们准确地完成了任务？ 尽管永远不能保证已经执行了所有可能的测试，但能肯定测试已经发现了错误(并且已修正了这些错误)。另外，如果已经制定了一个测试计

划，则可以进行检查以保证所有计划测试已被完成。

1.1 什么是软件测试

Glenford J. Myers (1979)[1]定义测试(Testing)是执行程序的过程，其目的是发现错误。IEEE 标准 610.12(1990)[2]给出了两个更为规范、约束的测试定义：

(1) 在特定的条件下运行系统或构件，观察或记录结果，对系统的某个方面做出评价。

(2) 分析某个软件项以发现现存的和要求的条件之差别(即错误)并评价此软件项的特性。

应该说，IEEE 610.12 标准的定义扩展了 Myers 的定义。IEEE 610.12 标准的定义(2)并不要求运行程序作为测试过程的一部分，静态验证的一些方法可作为测试手段。定义(1)被称为动态测试，而定义(2)被称为静态测试。软件是由文档、数据以及程序等工件组成，软件测试应该对于软件形成过程的文档、数据，以及程序进行测试。定义(2)的技术可以应用于非程序的工件。本书将介绍几种常见的静态测试，主要介绍软件的动态测试。

注意，英文 testing 和 test 在翻译成中文时，虽然都译为"测试"，但它们是有区别的。testing 是指一个过程，而 test 一般是执行测试的一次活动。在看到"测试"一词时，要根据上下文来判断是哪种意思。本书中的"测试"是指 testing。

软件测试是软件开发过程的重要组成部分，用来确认一个程序的品质或性能是否符合开发之前所提出的一些要求。软件测试是在软件投入运行前，对软件需求分析、设计规格说明和编码的最终评审(Review)，是软件质量保证的关键步骤。

软件测试是为了发现错误而执行程序的过程。软件测试在软件生命期中要横跨其中的两个阶段：

(1) 通常在编写出每一个模块之后就对它做必要的测试(称为单元测试)。编码和单元测试属于软件生存期中的同一个阶段。

(2) 在结束这个阶段后对软件系统还要进行各种综合测试，这是软件生存期的另一个独立阶段，即测试阶段。

注意，有资料表明，60%以上的软件错误并不是程序错误，而是软件需求和软件设计错误。错误的理解用户需求会导致开发技术完美、优良，但却不是正确的产品。因此，做好软件需求和软件设计阶段的质量保证工作也是非常重要的。

1.2 软件测试目的

Myers 这样来描述软件测试的目的："测试是程序的执行过程，目的在于发现错误；一个好的测试用例是指很可能找到迄今为止尚未发现的错误的用例。一个成功的测试是指发

现了至今尚未发现的错误的测试。”

Bill Hetzel[3]提出了测试目的不仅仅是为了发现软件缺陷与错误，而且也是对软件质量进行度量和评估，以提高软件的质量。

测试的目的是要以最少的人力、物力和时间找出软件中的各种错误与缺陷，通过修正各种错误与缺陷来提高软件质量，避免软件发布后由于潜在的软件缺陷和错误造成的隐患所带来的经济风险。同时，测试是以评价一个程序或者系统属性为目标的活动，测试是对软件质量的度量与评价，以验证软件的质量满足用户的需求的程度，为用户选择与接受软件提供有力的依据。

此外，通过分析错误产生的原因还可以帮助发现当前开发工作所采用的软件过程的缺陷，以便进行软件过程改进。同时通过对软件结果的分析整理，为风险评估提供信息，还可以修正软件开发规则，并以软件可靠性分析提供依据。当然，通过最终的验收测试，也可以证明软件满足了用户的需求，树立人们使用软件的信心。

软件质量可用几个方面来衡量：

(1) 在正确的时间用正确的方法做正确的事情(Doing the Right Things Right at the right time)。

(2) 符合一些应用标准的要求，比如不同国家中用户不同的操作习惯和要求，项目工程中的可维护性、可测试性等要求。

(3) 质量本身就是软件达到了最开始所设定的要求，而优美或精巧的表现技巧并不代表软件的高质量(Quality is defined as conformance to requirements, not as “goodness” or “elegance”)。

(4) 质量也代表着它符合客户需要(Quality also means “meet customer needs”)。在软件测试这个行业中，最重要的一件事就是从用户需求出发，从用户的角度去看产品，用户会怎么去使用这个产品，使用过程中会遇到什么样的问题。只有这些问题都解决了，软件产品的质量才可以说是得到了保证。

测试人员的总体目标是确保软件的质量，他们在软件开发过程中的任务是：寻找错误(Bug)，避免软件开发过程中的缺陷，衡量软件的品质，关注用户需求。

1.3 软件测试原理

上面讲到测试的目的是为了寻找软件的错误与缺陷、评估与提高软件质量。那么为了要达到测试目的，应遵循以下软件测试的基本原则。

- 所有的测试都应追溯到用户需求，软件测试的目标在于揭示错误。而最严重的错误(从用户角度来看)是那些导致程序无法满足需求的错误。
- 测试计划的制定应先于测试的执行。测试计划可以在需求模型一完成就开始，详细

的测试用例定义可以在设计模型被确定后立即开始，因此，所有测试可以在任何代码被产生前进行计划和设计。

- 帕累托法则适用于软件测试。简单而言，帕累托法则暗示着测试发现的错误中的80%很可能起源于程序模块中的20%。当然，问题在于如何孤立这些有疑点的模块并进行彻底的测试。
- 软件测试应从“小规模”开始，然后扩展到“大规模”。最初的测试通常把焦点放在单个程序模块上，进一步测试的焦点则转向在集成的模块簇中寻找错误，最后在整个系统中寻找错误。
- 完全测试是不可能的。即使一个大小适度的程序，其路径排列组合的数量也非常大，因此，在测试中不可能运行路径的每一种组合，然而，充分覆盖程序逻辑，并确保程序设计中使用的所有条件是有可能的。
- 要是测试更为有效，测试应由独立的第三方进行。“最佳效果”指最可能发现错误的测试(测试的主要目标)。创建系统的软件工程师并不是构造软件测试的最佳人选。从心理学的角度上来讲，软件分析和设计(包括编码)是建设性的工作。从开发者的观点来看，测试可以被看作是(从心理上来说)破坏性的。当测试开始的时候，就会存在一种微妙的、但确实存在着的、试图要“摧毁”软件工程师建立起来的东西的企图。所以开发者只是简单地设计和进行能够证明程序正确性的测试，而不是去尽量发现错误。独立测试组织的功能就是为了避免让开发者来进行测试时会引发固有问题。独立的测试可以消除可能存在的利益冲突。

1.4 软件测试过程

软件测试是一个复杂的过程，通常包括以下基本的测试活动：

(1) 拟定软件测试计划 (Planning)。

(2) 编制软件测试大纲 (Strategy)。

(3) 设计和生成测试用例。

(4) 实施测试。

(5) 分析测试结果。

1. 拟定软件测试计划

拟定软件测试计划就是确定主要的目标、测试范围、系统功能和非功能性需求、测试环境、测试自动控制、测试结果分析计划、问题解决方案与报告计划、测试重用计划、系统恢复计划、活动时间表、测试结束标准。

2. 编制软件测试大纲

软件测试大纲是软件测试的依据。它明确详尽地规定了在测试中针对系统的每一项功能或特性所必须完成的基本测试项和测试完成的标准。无论是自动测试还是手动测试，都必须满足测试大纲的要求。

3. 设计和生成测试用例

一般而言，测试用例是指为实施一次测试而向被测系统提供的输入数据、操作或各种环境设置以及被测系统的期望输出。测试用例控制着软件测试的执行过程，它是对测试大纲中每个测试项的进一步实例化。从工程实践的角度讲，设计测试用例的各种规则和策略有几条基本准则。

(1) 测试用例的代表性：能够代表各种合理和不合理的、合法的和非法的、边界和越界的，以及极限的输入数据、操作和环境设置等。

(2) 测试结果的可判定性：即测试执行结果的正确性是可判定的或可评估的。

(3) 测试结果的可再现性：即对同样的测试用例，系统的执行结果应当是相同的。

4. 实施测试

软件测试的实施阶段由一系列测试周期(Test Cycle)组成。在每个测试周期中，软件测试工程师将依据预先编制好的测试大纲和准备好的测试用例，通过执行被测软件，对其进行测试，即向被测软件输入数据或激发事件，以观察输出结果。

5. 分析测试结果

在执行软件测试的过程中，收集通过与未通过的测试用例。后者将触发纠错过程。测试与纠错通常是反复交替进行的。当使用专业测试人员时，测试与纠错甚至是平行进行的，从而压缩总的开发时间。测试结果分析可生成软件问题报告供有关人员参考或作进一步分析。

有的文章或书将第(1)至第(3)步骤合并在一起，通称为测试计划；也有的文章或书将第(2)和第(3)步骤合并在一起，称为测试策略。无论以什么方式合并，测试用例的设计和生成是主要的工作。本书将介绍的若干白盒测试技术与黑盒测试技术都是用来设计和生成测试用例的。

什么是测试用例？IEEE 610.12 标准测试用例的定义如下：

(1) 测试用例是(A)一组输入即运行前提条件，和为某特定的目标而生成的预期结果，例如：测试某一特定的程序路径或验证程序是否符合某特定需求。

(2) 测试用例是(B)一个文档，详细说明输入、期望输出，和为一测试项所准备一组的执行条件。

IEEE 610.12 标准的(A)是测试用例的实质，而(B)是测试用例的一种存在方式。简单地说，测试用例应由测试输入数据和与之对应的预期输出结果两部分组成。输入数据时可能需要附带一些条件。

白盒测试需要了解产品的内部工作，关注程序的结构和内部逻辑，根据程序的结构和内部逻辑设计用例。常用的白盒测试技术有：

- 基本路径测试(Basis path testing)，即为测验控制流路径设计测试用例。
- 条件测试(Condition testing)，即为测验每个条件的结果而设计测试用例。
- 数据流测试(Data flow testing)，即为测试数据元素定义和引用关系而设计的测试用例。

黑盒测试需要了解功能性的规格说明，关注对功能的需求，为测试系统的功能设计测试用例。常用的黑盒测试技术有：

- 边界值分析(Boundary value analysis)，即根据变量的边界值设计测试用例。
- 因果测试(casual-effect analysis)，即根据触发-响应和输入-输出的关系设计测试用例。
- 等价划分(equivalence partitioning)，即将输入、输出域划分成不相交的区域，根据这种划分设计测试用例。

1.5 软件测试类型

软件测试贯穿整个软件定义与开发整个期间。需求分析、概要设计、详细设计以及程序编码实现等各阶段所得到的文档，包括需求规格说明、概要设计规格说明、详细设计规格说明以及源程序，都是软件测试的对象。本节将从 4 个不同的角度讲述测试类型：开发阶段、测试技术、测试实施状态、测试实施主体。

1.5.1 按照开发阶段划分

软件测试按照开发阶段可划分为：单元测试、集成测试、确认测试、系统测试和验收测试。

1. 单元测试

单元测试完成对最小的软件设计单元——模块的验证工作。使用过程设计描述作为指南，对重要的控制路径进行测试以发现模块内的错误。测试的相关复杂度和发现的错误是由单元测试的约束范围来限定的。一般的过程如下：

(1) 为每一个软件组件设计单元测试。

(2) 评审每个单元测试以确保合适的覆盖。

(3) 执行单元测试。

(4) 更正发现的错误。

(5) 重新进行单元测试。

2. 集成测试

在所有的模块都已经完成单元测试之后，可能会遇到这样一个似乎很合理的问题："如果它们每一个都能单独工作得很好，那么为什么要怀疑把它们放在一起就不能正常工作呢？"当然，这个问题就在于"把它们如何放在一起？"——即接口的问题。数据可能在通过接口的时候丢失；一个模块可能对另外一个模块产生无法预料的副作用；当子函数被联到一起的时候，可能不能达到所期望的功能；在单个模块中可以接受的不精确性在联合起来之后可能会扩大到无法接受的程度；全局数据结构可能也会存在问题。集成测试是通过测试发现和接口有关的问题来构造程序结构的系统化技术，它的目标是利用通过了单元测试的模块，构造一个在设计中所描述的程序结构。

3. 确认测试

当集成测试结束的时候，软件就全部组装到一起了，接口错误已经被发现并修正了，而软件测试的最后一部分（确认测试）就可以开始了。确认可以通过多种方式来定义，但是，一个简单（虽然很粗糙）的定义是当软件可以按照用户合理的期望方式来工作的时候，确认测试就算成功。一个爱挑毛病的软件开发人员可能会提出抗议："谁或者什么来作为合理期望的裁定者呢？"合理期望在描述软件的所有用户可见的属性文档——软件需求规约中被定义。这个规约包含了标题为"确认标准"的一节内容，此信息就形成了确认测试方法的基础。

4. 系统测试

系统测试旨在测试属于整个系统的行为和错误的属性，而这些行为和错误不同于构件的属性。系统测试问题的例子包括：资源损失错误、吞吐量（Throughput）错误、性能错误、安全错误、恢复错误、事务同步错误（通常误命名为适时错误）。

5. 验收测试

如果软件是给一个客户开发的，则需要进行一系列的验收测试来保证客户对所有的需求都满意。接收测试是由最终用户而不是系统开发者来进行的，它的范围从非正式的"测试驱动"直到有计划的、系统化进行的系列测试。事实上，接收测试可以进行几个星期或者几个月，因此可以发现随着时间流逝可能会影响系统的累积错误。如果一个软件是给许多客户使用的，那么让每一个用户都进行正式的验收测试是不切实际的。大多数软件厂商使用一个被称作α测试和β测试的过程来发现那些似乎只有最终用户才能发现的错误。

1.5.2 按照测试技术划分

软件测试按照技术可划分为：白盒测试、黑盒测试及灰盒测试。

1. 白盒测试

白盒测试是指基于一个应用代码的内部逻辑知识(如全部代码、分支、路径、条件)来设计测试用例。测试退出条件是基于代码覆盖。由于能清楚地了解程序结构和处理过程并以此而进行测试，而被称为白盒测试。

2. 黑盒测试

黑盒测试是指不基于内部设计和代码的任何知识，而是基于需求和功能性，通过软件的外部表现来发现其缺陷和错误。黑盒测试法把测试对象看成一个黑盒子，不考虑程序内部结构和处理过程。

3. 灰盒测试

灰盒测试技术是一种有效的、介于白盒测试与黑盒测试之间的技术，它既关注程序运行时的外部表现，又注意程序内部高层逻辑结构。灰盒测试的优点是测试结果可以对应到程序的内部粗粒度路径，便于缺陷的定位、分析和解决。

软件测试方法的技术分类与软件开发过程相关联，单元测试一般应用白盒测试方法，集成测试应用灰盒测试方法，而系统测试和确认测试应用黑盒测试方法。

1.5.3 按照执行状态划分

软件测试按照执行状态可划分为静态测试和动态测试。

1. 静态测试

静态测试指不运行程序，而通过人工或利用自动检测工具对程序和文档进行分析与检查。静态测试技术又称为静态分析技术，是对软件中的需求说明书、设计说明书、程序源代码等进行非运行的检查。静态测试包括走查、审查、符号执行等。

2. 动态测试

动态测试指通过人工或利用工具运行程序进行检查，分析程序的执行状态和程序的外部表现。单元测试、集成测试、确认测试、系统测试、验收测试、白盒测试、黑盒测试及灰盒测试等是指动态测试。

本书中如没有特殊说明，测试均指动态测试，即测试需通过执行程序来完成。

1.5.4 按照执行主体划分

软件测试按照执行组织可划分为：开发方测试、用户测试、第三方测试。

1. 开发方测试

开发方测试通常也叫"验证测试"或"α 测试"。开发方通过检测和提供客观证据，证实软件的实现是否满足规定的需求。验证测试是在软件开发环境下，由开发者检测与证实软件的实现是否满足软件设计说明或软件需求说明的要求。主要是指在软件开发完成以后，开发方对要提交的软件进行全面的自我检查与验证，可以和软件的"系统测试"一并进行。

2. 用户测试

用户测试是指在用户的应用环境下，用户通过运行和使用软件，检测与核实软件实现是否符合自己预期的要求。通常情况用户测试不是指用户的"验收测试"，而是指用户的使用性测试，由用户找出软件的应用过程中发现的软件的缺陷与问题，并对使用质量进行评价。β 测试通常被看作是一种"用户测试"。β 测试主要把软件产品有计划地免费分发到目标市场，让用户大量使用，并评价、检查软件。通过用户各种方式的大量使用，来发现软件存在的问题与错误，把信息反馈给开发者修改。β 测试中厂商获取的信息，有助于软件产品的成功发布。

3. 第三方测试

第三方测试是指介于软件开发方和用户方之间的测试组织的测试。第三方测试也称为独立测试。软件质量工程强调开展独立验证和确认（Ⅳ&Ⅴ）活动。Ⅳ&Ⅴ是由在技术、管理和财务上与开发组织具有规定程度独立的组织执行验证和确认过程。软件第三方测试也就是由在技术、管理和财务上与开发方和用户方相对独立的组织进行的软件测试。一般情况下是在模拟用户真实应用的环境下，进行软件确认测试。

1.6 软件测试的注意事项(Tip)

测试人员应该按照软件测试的原则(Principle)开展测试活动：

(1) 应当把"尽早地和不断地进行软件测试"作为软件开发者的座右铭。

(2) 测试用例应由测试输入数据和与之对应的预期输出结果两部分组成。

(3) 程序员应避免检查自己的程序(注意不是指对程序的调试)。

(4) 在设计测试用例时，应当包括合理的输入条件和不合理的输入条件。不合理的输入条件是指异常的、临界的、可能引起问题异变的输入条件。

(5) 充分注意测试中的群集现象。经验表明，测试后程序残存的错误数目与该程序中以发现的错误数目或检错率成正比。应该对错误群集的程序段进行重点测试。

(6) 严格执行测试计划，排除测试的随意性。测试计划应包括所测软件的功能、输入和输出、测试内容、各项测试的进度安排、资源要求、测试资料、测试工具、测试用例的选择、测试的控制方法和过程、系统的组装方式、跟踪规则、调试规则、回归测试的规定以及评价标准等。

(7) 应当对每一个测试结果做全面的检查。

(8) 妥善保存测试计划、测试用例、出错统计和最终分析报告，为维护提供方便。

(9) 执行集成和确认测试。

(10) 更正发现的错误。

1.7 软件测试的现状和趋势与面临的挑战

在软件业较发达的国家，无论从投入的人力和时间上来看，软件测试都受到软件公司的极大重视。相比较而言，国内的软件测试还属于起步阶段，缺乏专业的第三方软件测试公司。

1.7.1 现状和趋势

下面从国际与国内两方面分析测试的现状和趋势。

1. 国际现状

美国著名软件质量分析师贺越明介绍了国外的情况，在软件业较发达的国家，软件测试不仅早已成为软件开发的一个有机组成部分，而且在整个软件开发的系统工程中占据着相当大的比重。以美国的软件开发和生产的平均资金投入为例，通常是："需求分析"和"规划确定"各占3%，"设计"占5%，"编程"占7%，"测试"占15%，"投产和维护"占67%。测试在软件开发中的地位，由此可见一斑。

与此同步的是，软件测试市场已成为软件产业中的一个独特市场。在美国硅谷地区，凡是软件开发企业或是设有软件开发部门的公司，都有专门的软件测试单位，其中软件测试人员的数量相当于软件开发工程师的3/4。在这些公司或部门中，负责软件测试的质量保证经理的职位与软件开发的主管往往是平行的。据了解，在软件产业发展较快的印度，软件测试在软件企业中同样拥有举足轻重的地位。

(1) 美国软件业依然保持着依靠软件产品统治软件业发展的传统。毋庸置疑，在以操作系统工程、数据库为代表的基础软件层次，美国几乎垄断了全球的软件市场。而如今全球

软件业发展到以网络互联、企业级应用、中间件为代表的新的时代，美国依然保持着行业领先位置。究其原因，当然是多方面的综合因素，但其中非常重要的一点是当今美国各行业的信息化水平非常高，残酷竞争更迫使企业在信息化方面加大投入，再加上美国企业信息化水平的基础本来就很好，从而使得对软件产品的需求非常明确，为软件企业如何开发最实用的产品提供了明确的定位。而目前中国企业的信息化程度虽然有了大幅度的提高，但是从深度和广度上还与美国等先进国家有着比较大的差距，中国的软件企业在产品和核心技术上要形成自己的产业难度很大。

(2) 对美国来说，软件工厂的概念已经完全形成，以CMM为标志的适应大规模生产的软件流程管理体系与质量管理体系已经非常完备，使软件行业真正成为制造业。几百名软件工程师有机地组织在一起为一个产品协同工作的事例已经非常普遍。而目前在中国，软件生产流程管理和质量管理都还处于相对初级的阶段，与美国等软件大国还有不小的差距，就是与印度相比，也是处于落后的态势。提高中国软件流程管理与质量管理的水平刻不容缓。

2. 国内现状

目前，国内软件测试市场表现实在令人担忧。中国市场中的软件开发公司比比皆是，但软件测试公司则如凤毛麟角。

为什么国内的软件测试市场会如此羸弱，到现在企业才开始关注呢？首先是因为企业对软件测试的重要性理解不够。很多人认为程序能试运行基本上就已经成功，没有必要成立专门的测试部门或设立测试岗位。

另一方面，软件开发企业在为软件开发支付费用后，就不希望再为软件的测试支付新的成本，而项目甲方则往往认为开发合格的软件是软件开发企业的责任。即使有些项目的开发方或委托方有意对软件进行第三方测试，也会考虑到在测试过程中往往需要软件开发商提供源代码，担心其知识产权遭到侵犯。这是软件测试市场无法长大的又一个重要原因。此外，软件开发企业严重缺乏专业测试力量也是因素之一。

3. 发展趋势

国际的测试领域已基本成熟，而中国的测试领域才刚刚开始。我们有很多的东西要去学习。如何更好地将软件项目管理和软件质量保障(软件测试)结合起来，让项目管理带动软件测试业的发展和成熟，应该是项目管理中不可缺少的一部分。

1.7.2 面临的挑战

当今快速发展的企业信息化进程导致软件测试面临复杂性、协调性和变化3个方面的技术挑战，同时对测试人员综合素质的要求也在不断提高。软件测试面临着技术发展的挑战与测试工程师素质的挑战。

1. 技术发展的挑战

在企业信息化的进程中，在线商务系统以及整个系统的可信任度已成为商务成功的核心要素之一，但是当今快速发展的电子商务技术环境使得整个电子商务系统高稳定性的获得比过去任何时候都更加困难，集中体现在3个方面：

1）复杂度

随着复杂的分布式应用技术的快速发展，电子商务应用的部署结构日益复杂，所涉及的协议和接口标准日益繁多，影响性能下降的原因也越来越多，通过 Internet 的访问无法预测，对应用部署前的性能评价要求越来越严格。

2）协调性

在当今多层分布式应用系统中，贯穿整个应用生命周期涉及大量异构组件或资源间的协调工作，这将会增加通信失败和错误发生的概率，所涉及各资源间的协调工作将极大地决定了应用的可信赖度。

3）变化

迫于市场的压力，电子商务应用开发周期变得越来越短，应用系统更新、升级日益频繁，在这种环境下必须特别关注整个应用的完整性和可靠性。

因此，能够满足复杂电子商务应用评测的企业级自动化测试平台必须能够保证：

- 分布式应用的内部互操作性。
- 全面支持 Java、EJB、RMI、CORBA、TUXEDO 等中间件技术。
- 准确定位应用失效或性能下降的原因。
- 提供可靠、高效的按照预定测试流程的自动化测试能力。
- 提供应用所涉及的不同组件、协调工作的整体评价指标。

2. 测试工程师素质的挑战

团队越规模越来越大，在一个测试团队中能否形成以核心人物为支柱的强有力的团队。你能否成为这个核心人物？这个核心人物人数的情况可能因为产品的测试组织结构不同而有所不同。

工程师的综合素质的高低体现在：责任心、综合技术素质、学习能力、解决问题的能力以及对软件业发展趋势的了解。

1.8 测试人员职业发展与具备的素质

测试人员拥有广阔的职业发展前景，测试人才的市场需求不断增多，相应的待遇也在不断增长。并且测试工作本身有助于提高分析解决问题的能力以及学习能力，使测试人员受

益匪浅。下面从两个角度来详细分析。

1.8.1 从测试工程师的市场角度来分析

“软件测试工程师”已成为人才需求中具有发展前景的四大亮点之一。随着中国IT行业的发展，产品的质量控制与质量管理正逐渐成为企业生存与发展的核心。从软件、硬件到系统集成，几乎每个大中型IT企业的产品在发布前期都需要大量的质量控制、测试和文档工作，而这些工作必须依靠拥有娴熟技术的专业软件人才来完成。软件测试工程师就是这样的一个企业重头角色。

在企业内部，软件测试工程师基本处于“双高”地位，即地位高、待遇高。可以说他们的职业前景非常广阔，从近期的企业人才需求和薪金水平来看，软件测试工程师的年工资有逐年上升的明显迹象。测试工程师这个职位必将成为IT就业的新亮点。

业内人士分析，该类职位的需求主要集中在沿海发达城市，其中北京和上海的需求量分别占去33%和29%。民营企业需求量最大，占19%；外商独资欧美类企业需求排列第二，占15%。

1.8.2 从测试工程师的自身素质提高的角度来看

无论从事什么样的职业，但是有两点在生活当中是永远不能抛弃掉的。

(1) 分析问题和解决问题的能力。生活在这空间里，我们身边不可能不发生任何问题，但是，问题发生了，需要做的是寻找合理的解决方法或者方案。这一点在测试领域尤为突出。在测试当中遇到问题了，其实，并不完全是软件的问题，有可能是系统环境、硬件环境或者有时是人为的原因，导致软件出了问题。现在需要做出分析判断，断定问题的原因，做出合理的处理。要么是软件质量的问题，要么是在使用软件中应该注意的问题。

(2) 学习的能力。软件在不断发展，起初软件只是在Windows平台运行，后来发展到了在Linux、UNIX、Solaris、AIX或者潜入到了芯片当中，但是你并没有这些知识，那么就需要在平时，或者在当时临时学习，这样的情况在测试领域很容易发生的。因此测试不单是一种工作，而且对从业人员的能力和思维都有很大提高。

1.9 总结

测试是要发现语义的或逻辑的错误，而不是要发现语法的或符号的错误。软件测试是为了确认软件做了所期望的事情，另一方面是确认软件以正确的方式来做了这个事件。软件测试不仅是在测试软件产品的本身，而且还包括软件开发的过程。如果一个软件产品开

发完成之后发现了很多问题,这说明此软件开发过程很可能是有缺陷的。因此软件测试是保证整个软件开发过程是高质量的。

1.10 参考文献

[1] Glenford J. Myers. The Art of Software Testing. John Wiley & Sons. 1 edition February 20,1979

[2] IEEE 610.12 标准-1990,IEEE standard glossary of software engineering terminology

[3] Bill Hetzel. The Complete Guide to Software Testing. Wiley. 2 edition. September 1993

1.11 思考与练习

1. 什么是软件测试?软件测试的目的是什么?
2. 如何理解:软件测试应从"小规模"开始,然后扩展到"大规模"?
3. 描述软件测试的过程。
4. 列举软件测试类型及各类型测试名称。
5. 查阅有关文献,说明测试面临的挑战。
6. 软件测试工程师应该具有什么素质?

1.12 进一步阅读

IEEE (1987). ANSI/IEEE Standard pp. 1008~1987, IEEE Standard for Software Unit Testing

IEEE (1990). IEEE Standard 610. pp. 12~1990, IEEE Standard Glossary of Software Engineering Terminology

Kaner, C., J. Falk, et al. (1999). Testing Comptuer Software. New York, Wiley Computer Publishing

McCabe, T. (1976). A Software Complexity Measure. IEEE Transactions on Software Engineering SE-2: pp. 308~320

R. Pressman. Software Engineering: A Practitioner's Approach. Boston: McGraw Hill, 2005

Boris Beizer. Software Testing Techniques, International Thomson Computer Press. 2nd edition. June 1990

Paul C. Jorgensen. Software Testing. A Craftsman's Approach, Second Edition. CRC Press, 2002

网站：PC Magzine, http://www.pcmag.com/encyclopedia_term/0,2542,t=test+case&i=52771,00.asp

1.13 课后作业

在讨论到"软件测试工程师的素质"时，有人认为软件测试工程师应具备以下素质。用你的理解，试分析一下，在下面所列举的素质中，哪些是软件测试工程师所特有的？哪些是软件工程师所共有的？

(1) 沟通能力：一名理想的测试者必须能够同测试涉及到的所有人进行沟通，具有与技术(开发者)和非技术人员(客户、管理人员)的交流能力。既要可以和用户谈得来，又能同开发人员说得上话，不幸的是这两类人没有共同语言。和用户谈话的重点必须放在系统可以正确地处理什么和不可以处理什么上。而和开发者谈相同的信息时，就必须将这些活重新组织以另一种方式表达出来，测试小组的成员必须能够同等地同用户和开发者沟通。

(2) 移情能力：系统开发的涉众(Stakeholder)，即和系统开发有关的所有人员，都处在一种既关心又担心的状态之中。用户担心将来使用一个不符合自己要求的系统，开发者则担心由于系统要求不正确而使他不得不重新开发整个系统，管理部门则担心这个系统突然崩溃而使它的声誉受损。测试者必须和每一类人打交道，因此需要测试小组的成员对他们每个人都具有足够的理解和同情，具备了这种能力，可以将测试人员与相关人员之间的冲突和对抗减少到最低程度。

(3) 技术能力：就总体而言，开发人员对那些不懂技术的人持一种轻视的态度。一旦测试小组的某个成员作出了一个错误的断定，那么他们的可信度就会立刻被传扬了出去。一个测试者必须既明白被测软件系统的概念又要会使用工程中的那些工具。要做到这一点需要有几年以上的编程经验，前期的开发经验可以帮助对软件开发过程有较深入的理解，从开发人员的角度正确的评价测试者，简化自动测试工具编程的学习曲线。熟悉编写程序的能力及开发环境，熟悉各种系统，比如：Linux、Solaris、AIX400、Windows 等，熟悉软件领域新技术的发展，比如：三层架构应用也相应地分为"前端接入，中间应用，后端数据库服务器"的三层模式的电子商务解决方案、数据库知识等。测试的产品不同，所要求的知识面也就有所不同了，总之，测试要求的知识面比较广。

(4) 自信心：开发者指责测试者出了错是常有的事，测试者必须对自己的观点有足够的自信心。

(5) 外交能力：当你告诉某人他出了错时，就必须使用一些外交方法。机智老练和外交手法有助于维护与开发人员的协作关系，测试者在告诉开发者他的软件有错误时，也同样

需要一定的外交手腕。如果采取的方法过于强硬，那么对测试者来说，在以后和开发部门的合作方面就相当于“赢了战争却输了战役”。

（6）幽默感：在遇到狡辩的情况下，一个幽默的批评将是很有帮助的。

（7）很强的记忆力：一个理想的测试者应该有能力将以前曾经遇到过的类似的错误从记忆深处挖掘出来，这一能力在测试过程中的价值是无法衡量的。因为许多新出现的问题和已经发现的问题相差无几。

（8）耐心：一些质量保证工作需要难以置信的耐心。有时需要花费惊人的时间去分离、识别和分派一个错误。这个工作是那些没有耐心的人无法完成的。

（9）怀疑精神：可以预料，开发者会尽他们最大的努力解释所有的错误。测式者必须听每个人的说明，但他必须保持怀疑直到他自己确认。

（10）自我督促：做测试工作很容易使你变得懒散。只有那些具有自我督促能力的人才能够使自己每天正常地工作。

（11）洞察力：一个好的测试工程师具有“测试是为了破坏”的观点，捕获用户观点的能力，强烈的质量追求，对细节的关注能力。应用高风险区的判断能力以便将有限的测试针对重点环节。

第2章

白 盒 测 试

白盒测试(White-box Testing)有时称为玻璃盒测试(Glass-box Testing),是一种基于源程序或代码的测试方法,分为静态和动态两种类型。静态方法是指按一定步骤直接检查源代码来发现错误,而不用生成测试用例并驱动被测程序运行来发现错误,也称为代码检查法;动态方法是指按一定步骤生成测试用例并驱动被测程序运行来发现错误。本章第一节和第二节介绍的动态方法有基本路径测试、条件测试、数据流测试及循环测试;第三节介绍的静态方法有桌面检查、代码审查及走查。

快速阅览:

什么是白盒测试? 基于源程序或代码结构与逻辑,生成测试用例以尽可能多地发现并修改源程序的错误。白盒测试分为静态和动态两种类型。静态测试方法有桌面检查、代码审查及走查,动态方法有基本路径测试、条件测试、数据流测试及循环测试。

由谁来负责白盒测试? 白盒测试一般由软件开发人员进行。在集成测试时如果用到白盒测试方法,一般由有经验的软件测试人员和软件开发人员共同完成集成测试的计划和执行。

为什么白盒测试如此重要? 白盒测试是一种主要的单元测试方法。如果基础单元质量不能保证,则会给后续测试工作、错误修正工作带来很多困难。

白盒测试步骤是什么? 白盒测试过程主要有5个步骤:根据源程序画程序图、生成测试用例、执行测试、分析覆盖标准、判定测试结果。

有哪些工件形成? 在一些情况下,会生成白盒测试计划、程序图或流图以及测试用例。白盒测试结果存档以便将来软件维护时使用。

如何确保准确地完成任务? 尽管永远不能保证已经执行了所有可能的白盒测试,但能肯定测试已经发现了错误(并且已修正了这些错误)。另外,如果已经制定了一个白盒测试计划,则可以通过检查来保证测试计划中的所有测试已被完成。

2.1 基本路径测试

基本路径测试是 Tom McCabe[1] 首先提出的一种白盒测试技术，基本路径测试给测试用例设计者提供方法来求出程序或过程设计中的逻辑复杂性测度，并使用该测度作为指南来定义执行路径的基本集。从该基本集导出的测试用例保证对程序中的每一条语句至少执行一次。

2.1.1 流图符号

在介绍基本路径方法之前，必须先介绍一种简单的控制流表示方法，即流图(Flow Graph)或程序图(Program Graph)。流图描述逻辑控制流，如图 2-1 所示。程序中每一种结构化构成元素都有一个相应的流图符号。

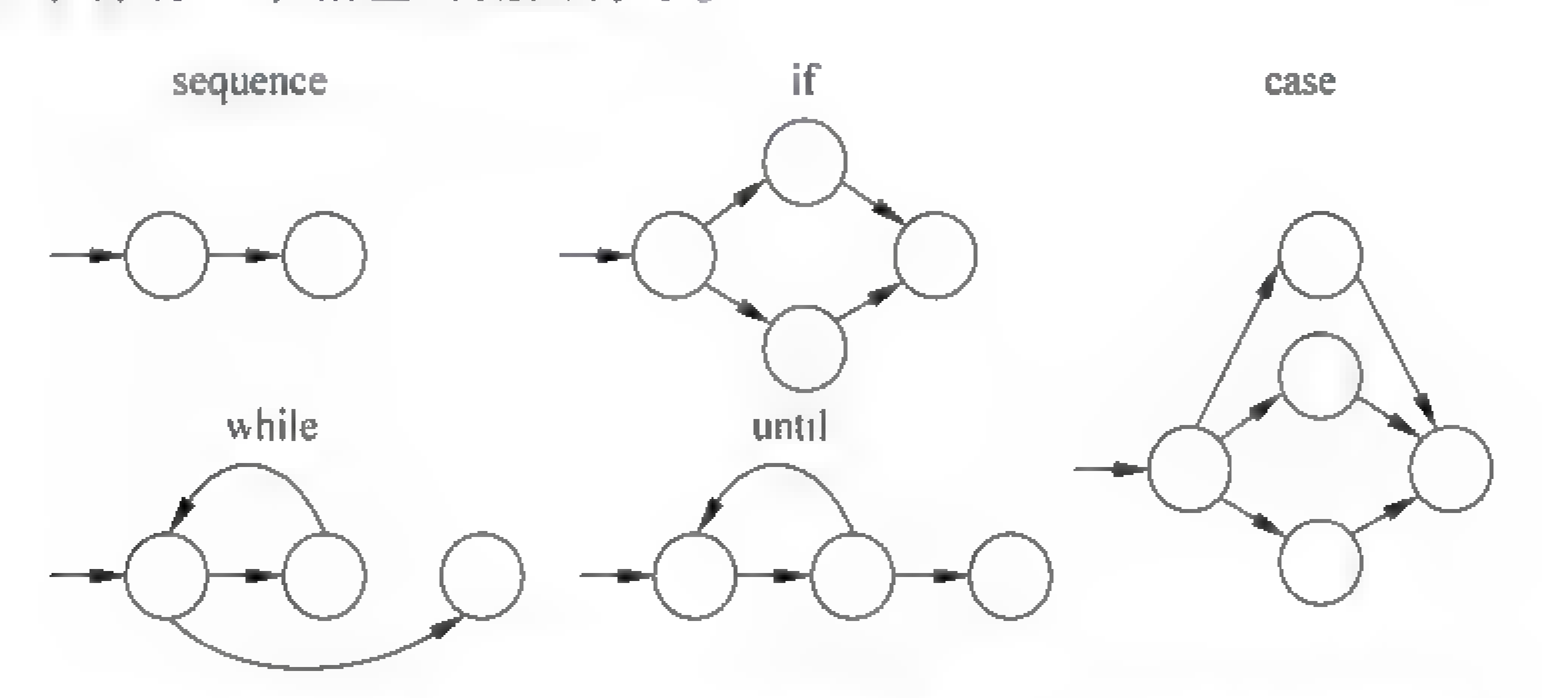


图 2-1　流图符号(每个圆表示一个或多个非分支 PDL 或源代码语句)

为了说明流图的用法，考察图 2-2(a)中的过程设计表示法。此处，流程图用来描述程序控制结构。图 2-2(b)将流程图映射到一个相应的流图(假设流程图的菱形决定框中不包含复合条件)。在图 2-2(b)中，每一个圆称为流图的节点，代表一个或多个语句。一个处理方框序列和一个菱形决策框可被映射为一个节点。流图中的箭头称为边或连接，代表控制流，类似于流程图中的箭头。一条边必须终止于一个节点，即使该节点并不代表任何语句(例如：参见 if-else-then 结构中的符号终结节点)。由边和节点限定的范围称为区域。计算区域时把图外部的区域算作一个区域。

程序设计中遇到复合条件时，生成的流图会变得稍微复杂。当条件语句中用到一个或多个布尔运算符(逻辑 OR、AND、NAND、NOR)时，就出现了复合条件。在图 2-3 中，将一段 PDL 语句翻译为流图，注意，为语句 IF a OR b 中的每一个 a 和 b 创建了一个独立的节点，包含一个条件的节点被称为判定节点，从每一个判定节点发出两条或多条边。任何过程

设计表示法都可被翻译成流图，图 2-3 显示了一个程序设计语言(Program Design Language, PDL)片段及其对应的流图，注意，这里对 PDL 语句进行了编号，并将相应的编号用于流图中。流图中编号后面括号里的内容是所关注的程序中的变量。

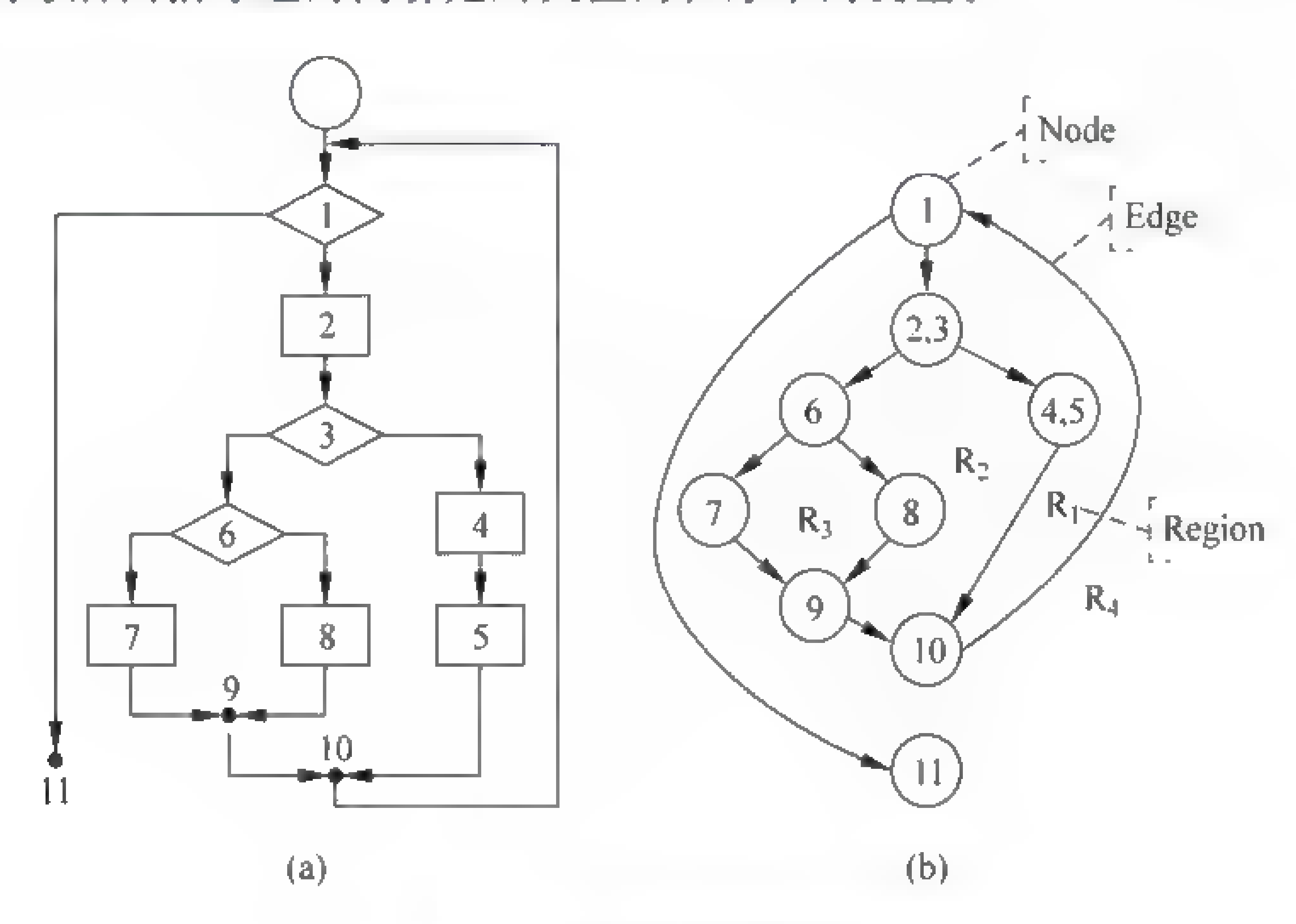


图 2-2　流程图和流图

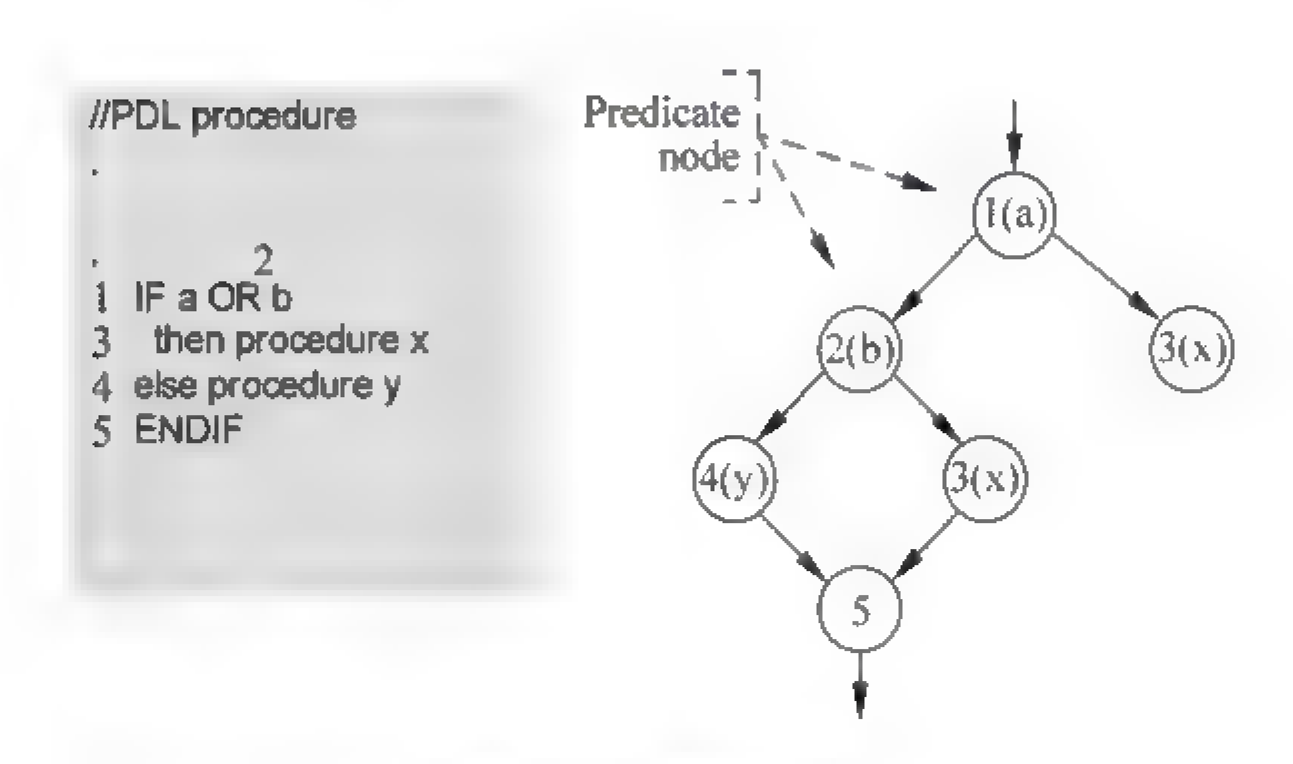


图 2-3　复合逻辑

2.1.2　独立程序路径

独立路径(也称为基本路径)是指在程序入口与出口之间的任一路径，其间不存在两条长度大于 2 的相同的子路径。下面给出流图和基本路径的定义。

一个流图定义为 $G=(V,E)$，其中 V 是顶点的集合，E 是有向边的集合。$V=\{v_b, v_e\}\cup D\cup S$，初始节点 v_b 的入度函数 indegree(v_b)的值为零，indegree(v_b)$=0$，结束节点 v_e 的出度函数 outdegree(v_e)的值为零，outdegree(v_e)$=0$，D 是原子二元判定条件的节点集，S 是

顺序的节点集，每一个节点表示一块连续的语句群；$E=D\times D\cup D\times S\cup S\times D$ 是有向边的集合。

路径 p 可以表示为一系列的顶点：$p=v_b,v_1,v_2,\cdots,v_n,v_e$，其中$\{v_b,v_i(i=1,\cdots,n),v_e\}\subseteq V$，$\{\langle v_b,v_1\rangle,\langle v_1,v_2\rangle,\cdots,\langle v_n,v_e\rangle\}\subseteq E$。路径 $p=v_b,v_1,v_2,\cdots,v_n,v_e$ 是一条基本路径，如果 p 不包含两个子序列的 s_1、s_2 使得 $s_1=s_2$，并且 $\text{length}(s_1)>1$ 和 $\text{length}(s_2)>1$。路径 p 的子序列 s_1，$\text{subseq}(s_1,p)$当且仅当$(\exists s_0)(\exists s_2)(s_0,s_1,s_2=p)$。

$$\neg(\exists s_1)(\exists s_2)(\text{subseq}(s_1,p)\wedge \text{subseq}(s_2,p)\wedge s_1=s_2\wedge |s_1|>1\wedge |s_2|>1)\text{(C1)}$$

例如，如图 2-2(b)中所示流图的如下 4 条路径为独立路径：

路径 1：1—11
路径 2：1—2—3—4—5—10—1—11
路径 3：1—2—3—6—8—9—10—1—11
路径 4：1—2—3—6—7—9—10—1—11

上面定义的路径 1、2、3 和 4 包含了如图 2-2(b)所示流图的一个基本集，简言之，如果能设计测试以便强迫运行这些路径(基本集)，那么程序中的每一条语句将至少被执行一次，每一个条件执行时都将分别取 true 和 false。应该注意到基本集并不唯一，实际上对给定的过程设计可导出任意数量的不同基本集。路径 1—2—3—4—5—10—1—2—3—6—8—9—10—1—11 不是独立路径，子序列 1—2—3 出现两次，不满足上述条件(C1)。

注意，有的书中定义“独立路径是指程序中至少引进一个新的处理语句集合或一个新条件的任一路径。采用流图的术语，即独立路径必须至少包含一条在定义路径之前未被用到的边[2]”。这种独立路径定义是不够精确的。比如先得到路径 2，然后考虑路径 1。路径 1 中的边{⟨1,11⟩}是路径 2 中的最后一条边，也就是说路径 1 相对路径 2 无新的路径增添。

2.1.3 环形复杂性

如何才能知道需要寻找多少条路径呢？对环形复杂性的计算结果为这个问题提供了答案。环形复杂性以图论为基础，为我们提供了非常有用的软件度量。

环形复杂性是一种为程序逻辑复杂性提供定量测度的软件度量。当该度量用于基本路径测试方法，计算所得的值给出了程序基本集的独立路径数量，这是为确保所有语句至少执行一次而必须进行测试数量的上界。可用如下 3 种方法之一来计算复杂性：

(1) 流图中区域的数量对应于环形的复杂性。

(2) 给定流图 G 的环形复杂性——CC(G)，定义为 $CC(G)=E-N+2$，E 是流图中边的数量，N 是流图节点数量。

(3) 给定流图 G 的环形复杂性——CC(G)，也可定义为 $CC(G)=P+1$，P 是流图 G 中判定节点的数量。

再回到图2-2(b)，可采用上述任意一种算法来计算环形复杂性。

(1) 流图有4个区域。

(2) CC(G)=11条边-9个节点+2=4。

(3) CC(G)=3个判定节点+1=4。

因此，图2-2(b)的环形复杂性是4。更重要的是CC(G)的值提供了组成基本集的独立路径的上界，并由此得出覆盖所有程序语句所需的测试设计数量的上界。

2.1.4 导出测试用例

基本路径测试方法可用于过程设计或源代码。本节介绍利用基本路径测试产生测试用例的一系列步骤，将用图2-4中PDL所描述的“清理列表”过程阐明测试用例设计方法中的各个步骤。

```
Procedure purge (var L: list)
     var p,q: …//define p,q
     begin
(1)  p := FIRST(L);
(2)        begin while p <> END(L) do
(3)           q := next(p,L);
(4)           begin while q <> END(L) do
(5)              if Aq = Ap then
(6)                 delete (Aq,L)
(7)              else q := next(q,L);
(8)           end
(9)        p := next(p,L)
(10)     end;
       end;
```

图2-4 PDL所描述的“清理列表”过程

(1) 以设计或代码为基础，画出相应的流图。使用符号和2.1.1节中的构造规则创建一个流图，参考图2-4中“求平均值”的PDL。创建流图时，要对将被映射为流图节点的PDL语句进行标号(1)～(10)，图2-5显示了对应的流图。

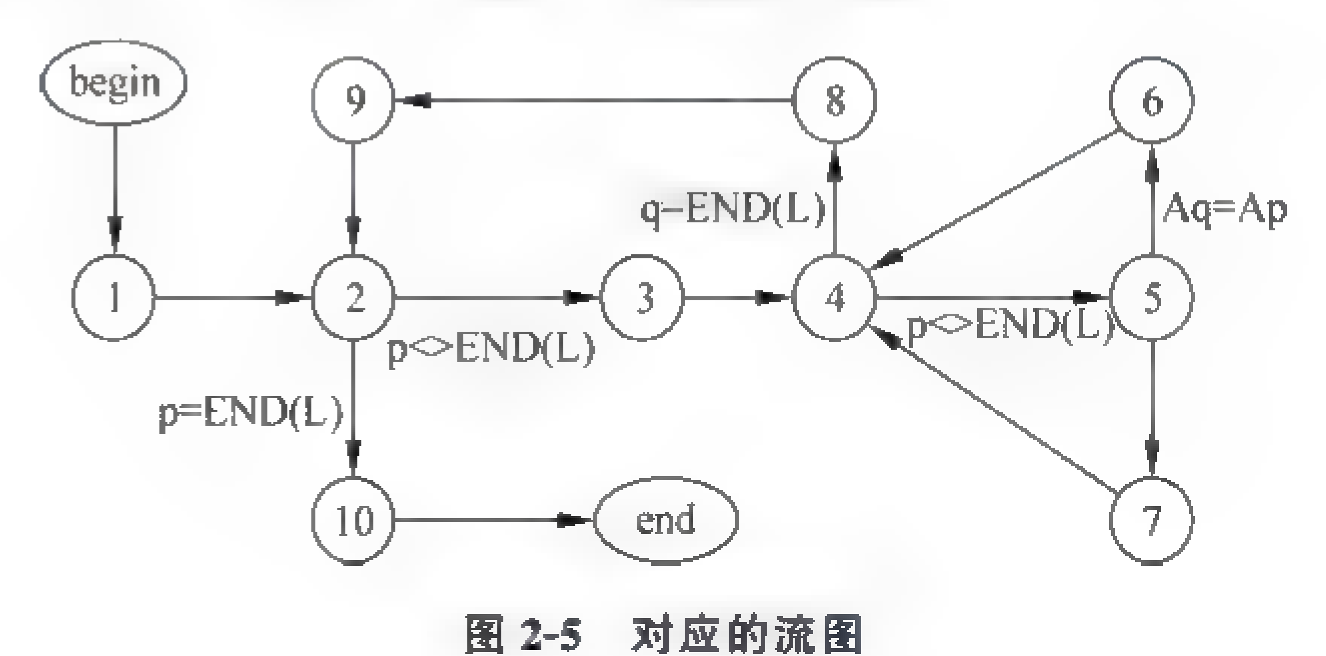


图2-5 对应的流图

(2) 确定结果流图的环形复杂性。可采用上一节中的任意一种算法来计算环形复杂性——CC (G)。计算 CC (G)并不一定要画出流图，计算 PDL 中的所有原子条件语句数量(while 及 if 条件语句)，然后加 1 即可得到环形复杂性。应该注意，如果是复合条件语句，需求出原子条件语句数量。对于图 2-5 中的流图，CC(G)的计算方法有以下三种：

CC(G) = 4 个区域
CC(G) = 14 条边 − 12 个节点 + 2 = 4
CC(G) = 3 个判定节点 + 1 = 4

(3) 确定线性独立的路径的一个基本集。CC (G)的值提供了程序控制结构中线性独立的路径的数量，通常在导出测试用例时，识别判定节点是很有必要的。在本例中，节点 2、4 和 5 是判定节点。在求平均值的过程中，指定 4 条路径：

路径 1：1—2—10
路径 2：1—2—3—4—8—9—2—10
路径 3：1—2—3—4—5—6—4—8—9—2—10
路径 4：1—2—3—4—5—7—4—8—9—2—10

(4) 准备测试用例，强制执行基本集中每条路径。测试人员可选择数据以便在测试每条路径时适当设置判定节点的条件。满足上述基本集的测试用例如下：

路径 1 测试用例：
  输入条件：L=( )，即列表为空
  期望结果：L=( )，无列表清理处理
路径 2 测试用例：
  输入条件：L=(Ap)
  期望结果：L=(Ap)
路径 3 测试用例：
  输入条件：L=(Ap Aq)且 Ap = Aq
  期望结果：L=(Ap)，从列表中清理 Aq
路径 4 测试用例：
  输入条件：L=(Ap Aq)且 Ap <> Aq
  期望结果：L=(Ap Aq)

执行每个测试用例，并和期望值比较，一旦完成所有测试用例，测试者就可以确定在程序中的所有语句至少被执行一次。重要的是，某些独立路径(如，例子中的路径 1)不能以独立的方式被测试，即穿越路径所需的数据组合不能形成程序的正常流，在这种情况下，这些路径必须作为另一个路径测试的一部分来进行测试。

2.1.5 图矩阵法

导出流图和决定基本测试路径的过程都需要机械化或自动化手段来支持。为了开发辅助基本路径测试的软件工具，一种称为图矩阵(Graph Matrix)的数据结构很有用。图矩阵是一个正方形矩阵，其大小(即列数和行数)等于流图的节点数。每列和每行都对应于标识

的节点，矩阵项对应于节点间的连接（边），图 2-6 显示了一个简单的流图及其对应的图矩阵[3]。

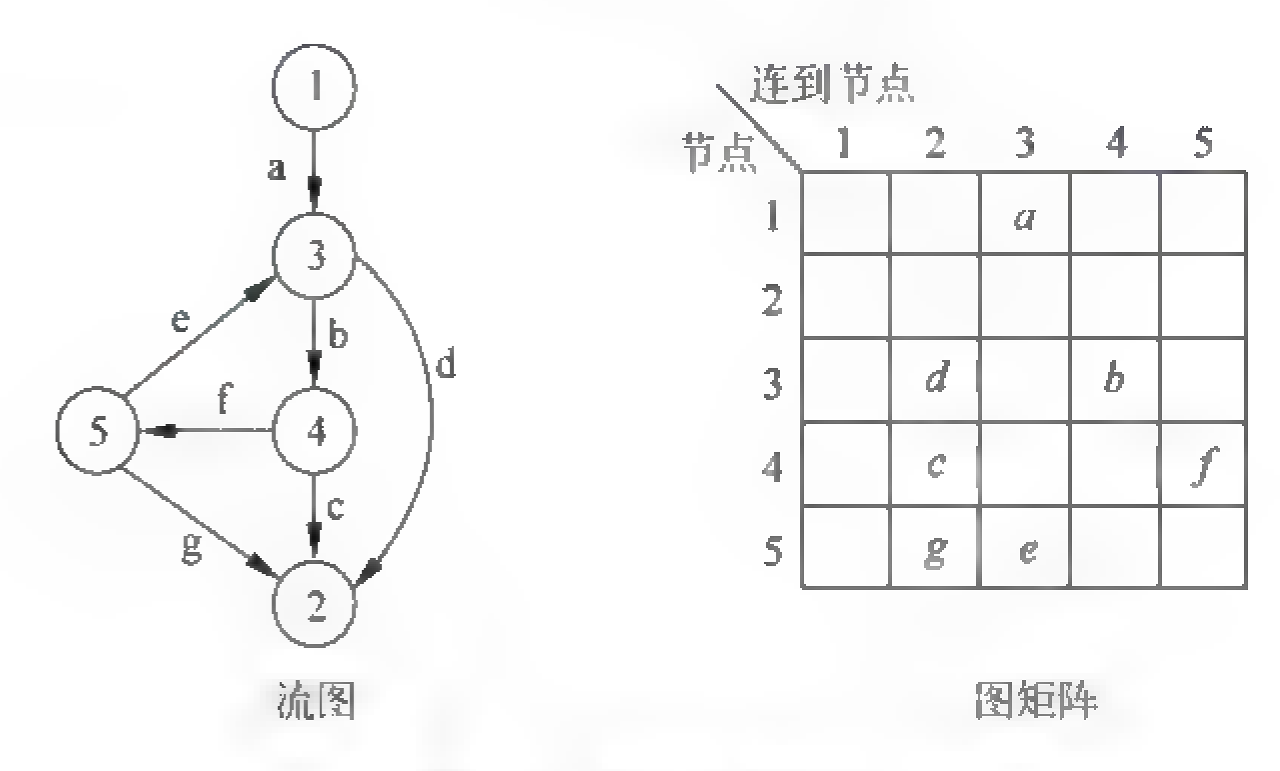


图 2-6 流图和图矩阵

在图 2-6 中，流图的节点以数字标识，边以字母标识，矩阵中的字母项对应于节点间的连接，例如，边 *b* 连接节点 3 和节点 4。这里，图矩阵只是流图的表格表示，然而，对每个矩阵项加入连接权值（Link Weight），图矩阵就可以用于在测试中评估程序的控制结构，连接权值为控制流提供了另外的信息。在最简单情况下，连接权值是 1（存在连接）或 0（不存在连接），但是，连接权值可以赋予更有趣的属性：

- 执行连接（边）的概率。
- 穿越连接的处理时间。
- 穿越连接时所需的内存。
- 穿越连接时所需的资源。

举例来说，用最简单的权值（0 或 1）来标识连接，如图 2-6 所示的图矩阵可重画为图 2-7。字母替换为 1，表示存在边（为清晰起见，没有画出 0），这种形式的图矩阵称为连接矩阵（Connection Matrix）。

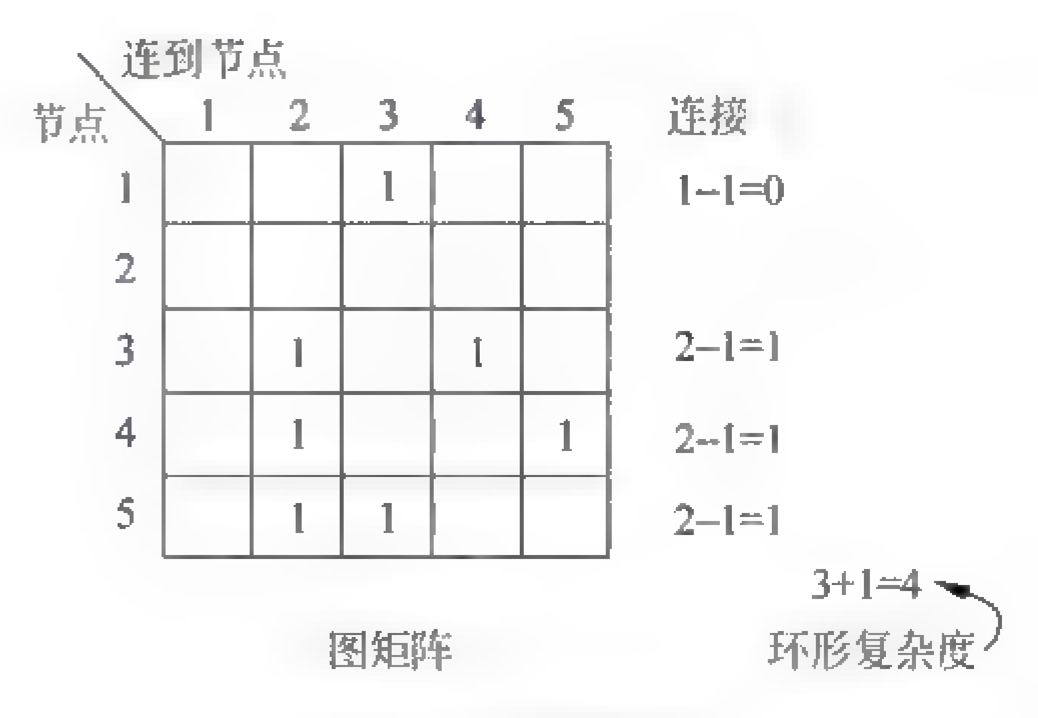


图 2-7 连接矩阵

在图2.7中，含两个或两个以上项的行，表示判定节点，所以，右边所示的算术计算就提供了另一种环形复杂性计算(参考2.1.3节)的方法。Beizer提供了可用于图矩阵的其他数学算法的处理，利用这些技术，设计测试用例的分析就可以自动化或部分自动化[3]。

2.2 控制结构测试

2.1节所描述的基本路径测试技术是控制结构测试技术之一。尽管基本路径测试简单高效，但是如果只用这一种方法，并不能充分地保证程序的质量。本节讨论控制结构测试的其他变种，这些测试覆盖提高了白盒测试的质量。

2.2.1 条件测试

条件测试是检查程序模块中所包含逻辑条件的测试用例设计方法。一个简单条件是指一个布尔变量或一个可能带有NOT(¬)操作符的关系表达式。关系表达式的形式如下：

$$E_1 \text{＜关系操作符＞} E_2$$

其中E_1和E_2是算术表达式，而＜关系操作符＞是下列之一：$<$、$\leq$、$=$、$\neq$($\neg=$)、$>$或$\geq$。复杂条件由两个或多个简单条件、布尔操作符和括弧组成。假定可用于复杂条件的布尔操作符包括OR 、AND(&)和NOT(¬)，不含关系表达式的条件称为布尔表达式。

所以条件的成分类型包括布尔操作符、布尔变量、一对布尔括弧(括住简单或复杂条件)、关系操作符或算术表达式。如果条件不正确，则至少有一个条件成分不正确，所以条件的错误类型如下：

- 布尔操作符错误(遗漏布尔操作符、布尔操作符多余或布尔操作符不正确)。
- 布尔变量错误。
- 布尔括弧错误。
- 关系操作符错误。
- 算术表达式错误。

条件测试方法注重于测试程序中的条件。本节后面讨论的条件测试策略主要有两个优点，首先，一个条件测试的覆盖率度量是简单的；其次，程序的各个条件测试覆盖率为产生另外的程序测试提供了指导。

条件测试的目的是测试程序条件的错误和程序的其他错误。如果程序P的测试集能够有效地检测P中的条件错误，则该测试集可能也会有效地检测P中的其他错误，此外，如果测试策略对检测条件错误有效，则它也可能有效地检测程序错误。

前面已经提出了几个条件测试策略。分支测试可能是最简单的条件测试策略，对于复合条件C，C的真分支和假分支以及C中的每个简单条件都需要至少执行一次[4]。

域测试(Domain testing)[5]要求从关系表达式中导出 3 个或 4 个测试用例，关系表达式的形式如下：

$$E_1 < 关系操作符 > E_2$$

需要 3 个测试用例分别用于计算 E_1 的值是大于(>)、等于(=)或小于(<)E_2 的值[6]。如果<关系操作符>错误，而 E_1 和 E_2 正确，则这 3 个测试用例能够发现关系操作符的错误。为了发现 E_1 和 E_2 的错误，计算 E_1 小于或大于 E_2 的测试用例应使两个值间的差别尽可能小。

有 n 个变量的布尔表达式需要 2^n 个可能的测试用例($n>0$)。这种策略可以发现布尔操作符、变量和括弧的错误，但是只有在 n 很小时实用。

也可以生成布尔表达式的敏感错误测试[7][8]。对于有 n 个布尔变量($n>0$)的单纯型布尔表达式(每个布尔变量只出现一次的布尔表达式)，可以很容易地产生测试数小于 2^n 的测试集，该测试集能够发现多个布尔操作符错误和其他错误。

Tai[9]建议在上述技术之上建立条件测试策略。称为 BRO(Branch and Relational Operator)的测试技术。假设在一个条件中所有的布尔变量和关系表达式只出现一次而且没有公共变量，BRO 保证能发现该条件中的分支(布尔)操作符和关系操作符错误。BRO 策略利用某条件 C 的条件约束。有 n 个简单条件的条件 C 的条件约束定义为(D_1, D_2, …, D_n)，其中 D_i($0<i\leqslant n$)表示条件 C 中第 i 个简单条件的输出约束。如果在执行条件 C 的过程中，C 的每个简单条件的输出都满足 D 中对应的约束，则称条件 C 的条件约束 D 由 C 的执行所覆盖。

对于布尔变量 B，B 输出的约束指定 B 必须是真(t)或假(f)。类似地，对于关系表达式符号<、=、>用于指定表达式输出的约束。

作为简单的例子，考虑条件 C_1，有：

C_1：B_1 & B_2

其中 B_1 和 B_2 是布尔变量。C_1 的条件约束式如(D_1, D_2)，其中 D_1 和 D_2 是 t 或 f，值(t, f)是 C_1 的一个条件约束，由使 B_1 为真使 B_2 为假的测试所覆盖。BRO 测试策略要求约束集{(t,t),(f,t),(t,f)}由 C_1 的执行所覆盖，如果 C_1 由于布尔操作符的错误而不正确，至少有一个约束强制 C_1 失败。

作为第二个例子，考虑条件 C_2，有：

C_2：B_1 & ($E_3 = E_4$)

其中 B_1 是布尔表达式，而 E_3 和 E_4 是算术表达式。C_2 的条件约束形式如(D_1, D_2)，其中 D_1 是 t 或 f，D_2 是<、=或>。除了 C_2 的第二个简单条件是关系表达式以外，C_2 和 C_1 相同，所以可以修改 C_1 的约束集{(t,t),(f,t),(t,f)}，得到 C_2 的约束集，注意($E_3 = E_4$)的 t 意味着=，而($E_3 = E_4$)的 f 意味着>或<。分别用(t,=)和(f,=)替换(t,t)和(f,t)，并用(t,<)和(t,>)替换(t,f)，就得到 C_2 的约束集{(t,=),(f,=),(t,<),(t,>)}。上述条件约束集的覆盖率将保证检测 C_2 的布尔和关系操作符的错误。

作为第三个例子,考虑条件 C_3,有:

C_3:$(E_1>E_2)\&(E_3=E_4)$

其中 E_1、E_2、E_3 和 E_4 是算术表达式。C_3 的条件约束形式如(D_1,D_2),其中 D_1 和 D_2 是<、=或>。除了 C_3 的第一个简单条件是关系表达式以外,C_3 和 C_2 相同,所以可以修改 C_2 的约束集得到 C_3 的约束集,结果为

{(>,=),(=,=),(<,=),(>,>),(>,<)}

上述条件约束集能够保证检测 C3 的关系操作符的错误。

2.2.2 数据流测试

数据流测试方法按照程序中的变量定义和使用的位置来选择程序的测试路径。已经有不少关于数据流测试策略的研究[10][11][12]。为了说明数据流测试方法,假设程序的每条语句都赋予了唯一的语句号,而且每个函数都不改变其参数和全局变量。对于语句号为 S 的语句:

def(S)={x|语句 S 包含 x 的定义}

use(S)={x|语句 S 包含 x 的使用}

如果存在从 S 到 S′的路径,并且该路径不含 x 的其他定义,则称变量 x 在语句 S 处的定义在语句 S′仍有效或称为定义清纯(Def-Clear)。变量 x 的"定义使用关联"(du-association)形式如[x,S,S′],其中 S 和 S′是语句号,x 在 def(S)和 use(S′)中,而且语句 S 定义的 X 在语句 S′有效。图 2-8 中变量 x 的 du-关联为[x,1,4],其相应 du-路径为(1,2,4)和(1,3,4)。如果语句 S 是 if 或循环语句,def(S)集为空,而 use(S)集取决于 S 的条件。

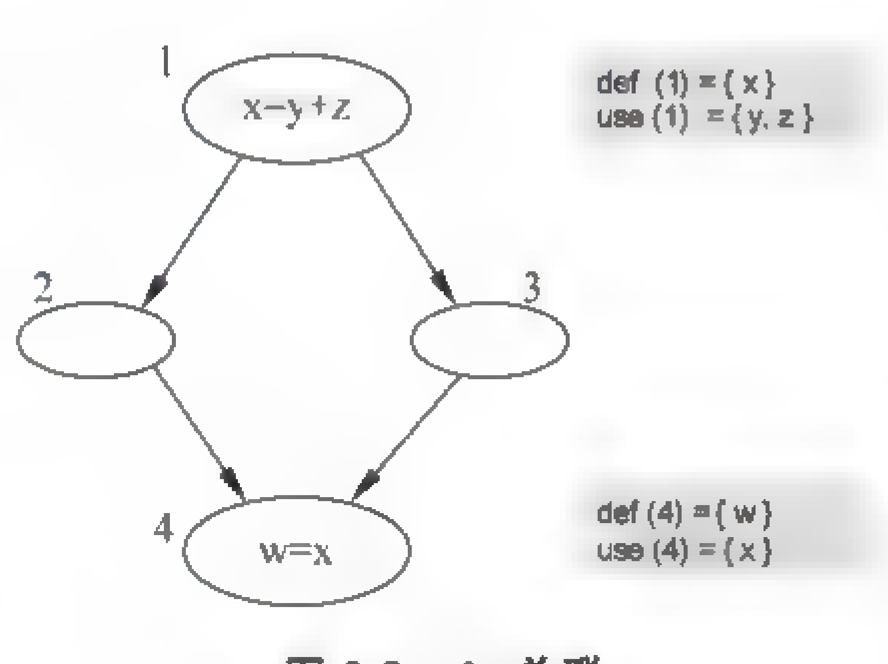


图 2-8 du-关联

一个变量的使用可以是计算使用(c-use)或断言使用(p-use)。一个变量的 du 链是一条路径,这条路径是从变量的定义到变量的使用之间定义清纯(definition clear),也就是说,无任何重定义。对于 c-use 来说,du 链是从含有定义语句到含有计算使用语句之间的路径。对于 p-use 来说,du 链是从含有定义语句到包含两个执行分支断言使用语句之间的路径。选择程序的测试路径是基于变量的 du 链及测试数据的适当标准而进行的(比如说,all-use)。图 2-9 中的例子展示了过程内部 c-use 和 p-use 的 du 链:[1,(2,3)],[1,(2,5)],[1,3],[1,5]。

图 2-10 中的例子展示了过程间 c-use 和 p-use 的 du 链。对于过程 du 链:[1,(5,6)],[1,(5,8)],[1,7]。

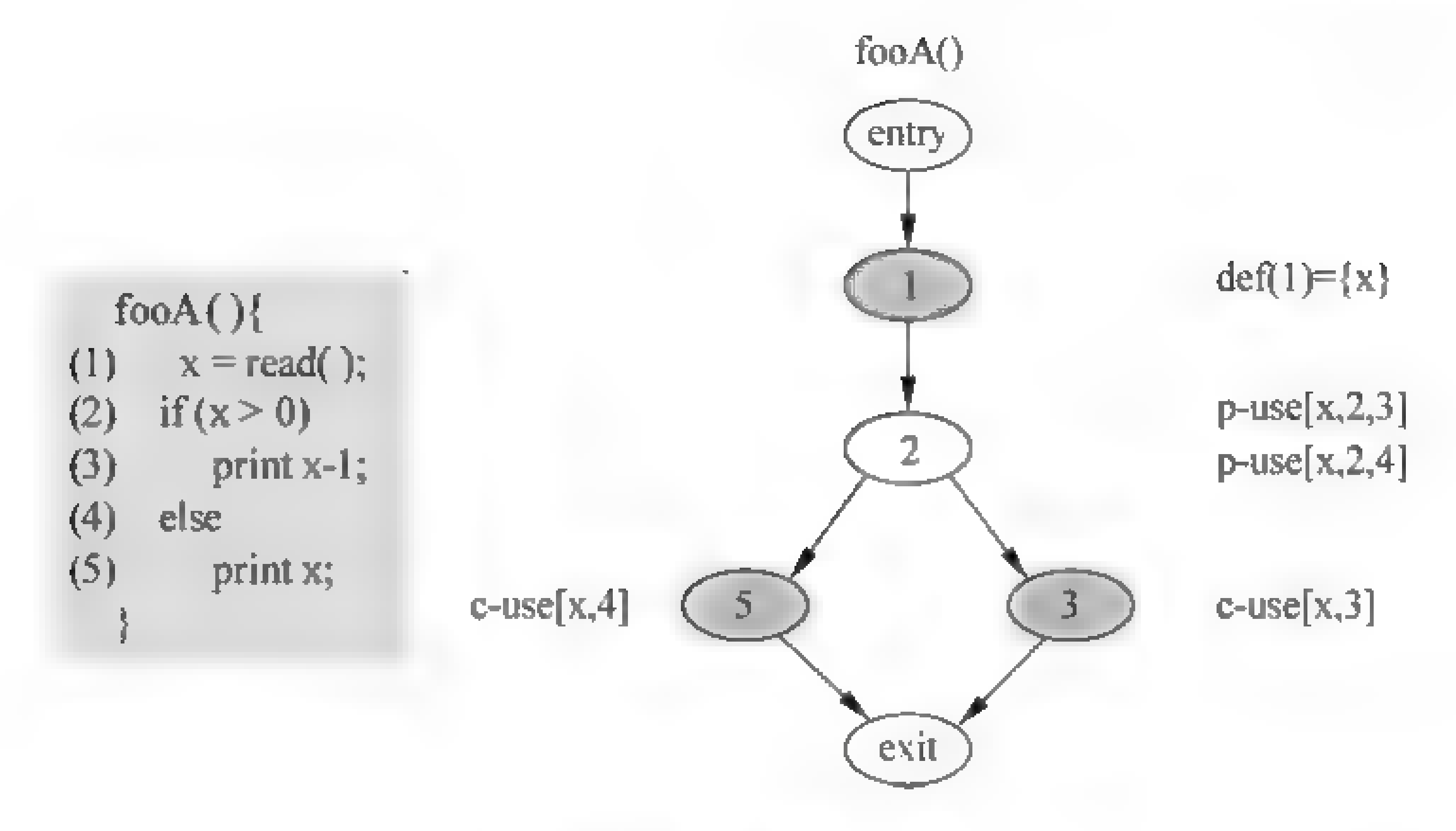


图 2-9　过程内部 du-链

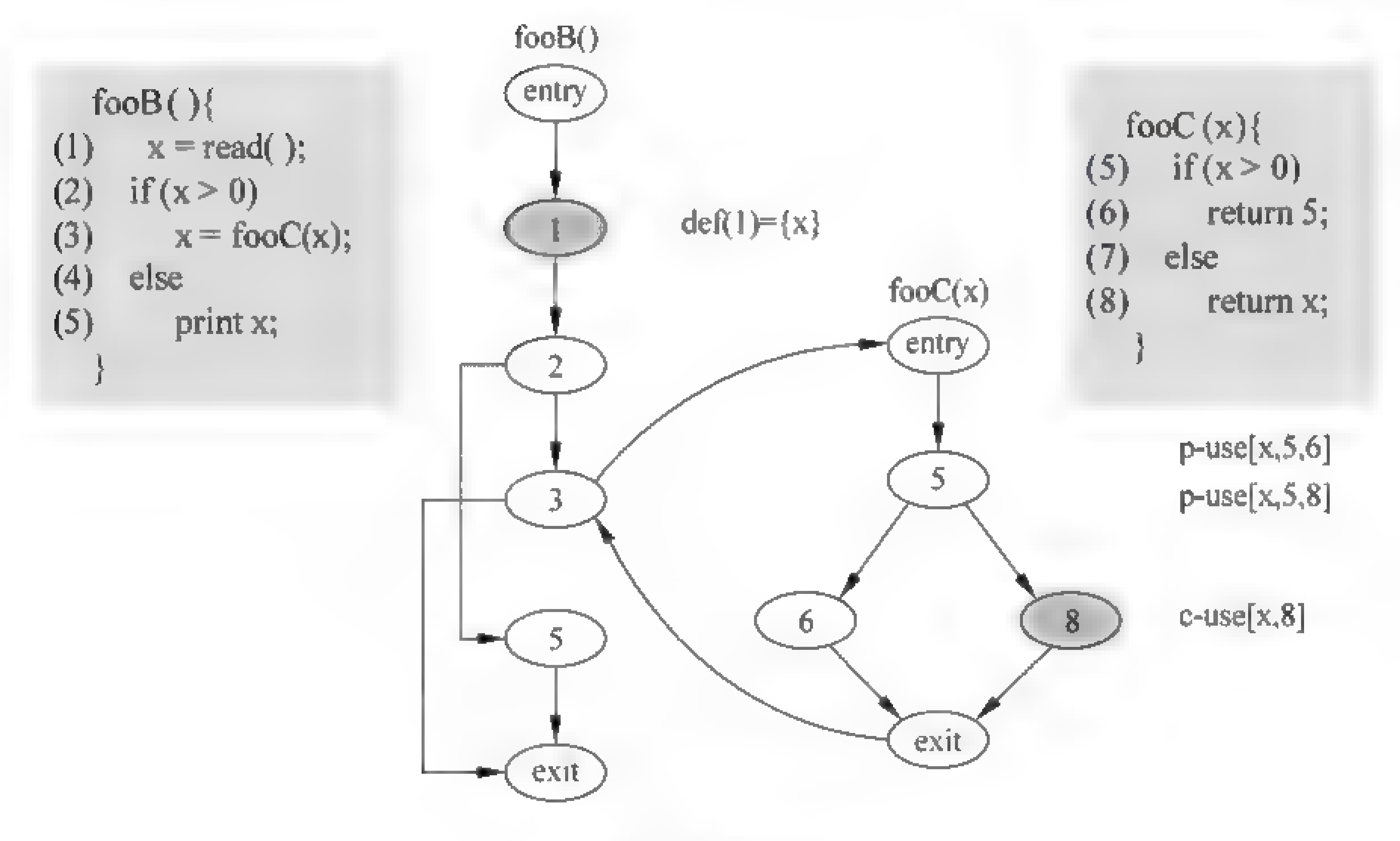


图 2-10　过程间 du-链

数据流测试方法生成测试用例的步骤：

(1) 为每个变量确定“定义—使用链”。

(2) 确定测试标准，比如说，取全部定义、取全部使用或取全部路径等。

(3) 设计测试用例来符合测试的标准。

一种简单的数据流测试策略是要求覆盖每个 du 链至少一次，这种策略称为 du 测试策略。已经证明 du 测试并不能保证覆盖程序的所有分支，但是，du 测试不覆盖某个分支仅仅在于如下之类的情况：if then else 中的 then 没有定义变量，而且不存在 else 部分。这种情况下，if 语句的 else 分支并不需要由 du 测试覆盖。

数据流测试策略可用于为包含嵌套 if 和循环语句的程序选择测试路径，为此，考虑使用 du 测试为如下的 PDL 选择测试路径：

```
proc x
(1)     B1; //define X at end of B1
(2)     do while C1
(3)         if C2
(4)         then
(5)             if C4
(6)                 then B4; //use x at the beginning of B4,define x at end of B4
(7)                 else B5; //use x at the beginning of B5,define x at end of B5
(8)             endif;
(9)         else
(10)            if C3
(11)                then B2; //use x at the beginning of B2,define x at end of B2
(12)                else B3; //use x at the beginning of B3,define x at end of B3
(13)            endif;
(14)        endif;
(15)    enddo;
(16)    B6;
end proc;
```

为了用 du 测试选择控制流图的测试路径，需要知道 PDL 条件或程序块中的变量定义和使用。假设变量 x 定义在块 B1、B2、B3、B4 和 B5 的最后一条语句之中，并在块 B2、B3、B4、B5 和 B6 的第一条语句中使用。du 测试策略要求执行从每个 $B_i(0<i\leqslant 5)$ 到 $B_j(1<j\leqslant 6)$ 的最短路径(这样的测试也覆盖了条件 C_1、C_2、C_3 和 C_4 中使用的变量 x)。尽管有 25 条 x 的 du 链，只需 5 条路径覆盖这些 du 链。原因在于可用 5 条从 $B_i(0<i\leqslant 5)$ 到 B_6 的路径覆盖 x 的链，而这 5 条链包含循环的迭代就可以覆盖其他的 du 链。

du—路径 1：(1)—(2)—(16)
du—路径 2：(1)—(2)—(3)—(4)—(5)—(6)—(8)—(14)—(15)—(16)
du—路径 3：(1)—(2)—(3)—(4)—(5)—(7)—(8)—(14)—(15)—(16)
du—路径 4：(1)—(2)—(3)—(9)—(10)—(11)—(13)—(14)—(15)—(16)
du—路径 5：(1)—(2)—(3)—(9)—(10)—(12)—(13)—(14)—(15)—(16)

注意如果要用分支测试策略为上述的 PDL 选择测试路径，并不需要另外的信息。为了选择 BRO 测试的路径，只需知道每个条件和块的结构(选择程序的路径之后，需要决定该路径是否实用于该程序，即是否存在执行该路径的至少一个输入)。

由于变量的定义和使用，程序中的语句都彼此相关，所以数据流测试方法能够有效地发现错误，但是，数据流测试的覆盖率度量和路径选择比条件测试更为困难。

图 2-11 总结了 6 种标准之间的涵盖关系。其中 all-du-paths、all-defs 和 all-uses 是基于数据流测试方法设计测试用例的标准。all-defs 包含了每一个定义到某一使用，all-uses 包含了每一个定义到每一个使用，all-du-paths 包含了从每一个定义到每一个使用的所有路

径 du-paths。程序或过程中的所有路径涵盖 du-paths。这一点很明显；du-paths 涵盖 all-uses，all-uses 要求包含了每一个定义到每一个使用的路径，在图 2-8 中，从[1,2,4]或[1,2,3]取任一个都满足 all-uses 标准，而 du-paths 标准两者都需取；按 all-uses 和 all-defs 定义，容易理解 all-uses 涵盖 all-defs，all-uses 要求覆盖从每一个定义到每一个使用的路径而 all-defs 只覆盖从每一个定义到某个使用的路径。

用图 2-12(a)和(b)说明 all-uses、all-edges 及 all-nodes 的涵盖关系。在图 2-12(a)中按 all-uses 标准要为变量 x 的 du 链[1,2,3,4]及变量 y 的 du 链[2,3]生成两个测试用例，而 all-edges 只生成 entry-1-2-3-4-exit 一个测试用例便满足标准，因为这个测试用例覆盖所有的边。在图 2-12(b)中按 all-nodes 只生成 entry-1-2-3-exit 一个测试用例便满足标准，而 all-edges 还要求生成 entry-1-2-3-1-2-3-exit 测试用例。

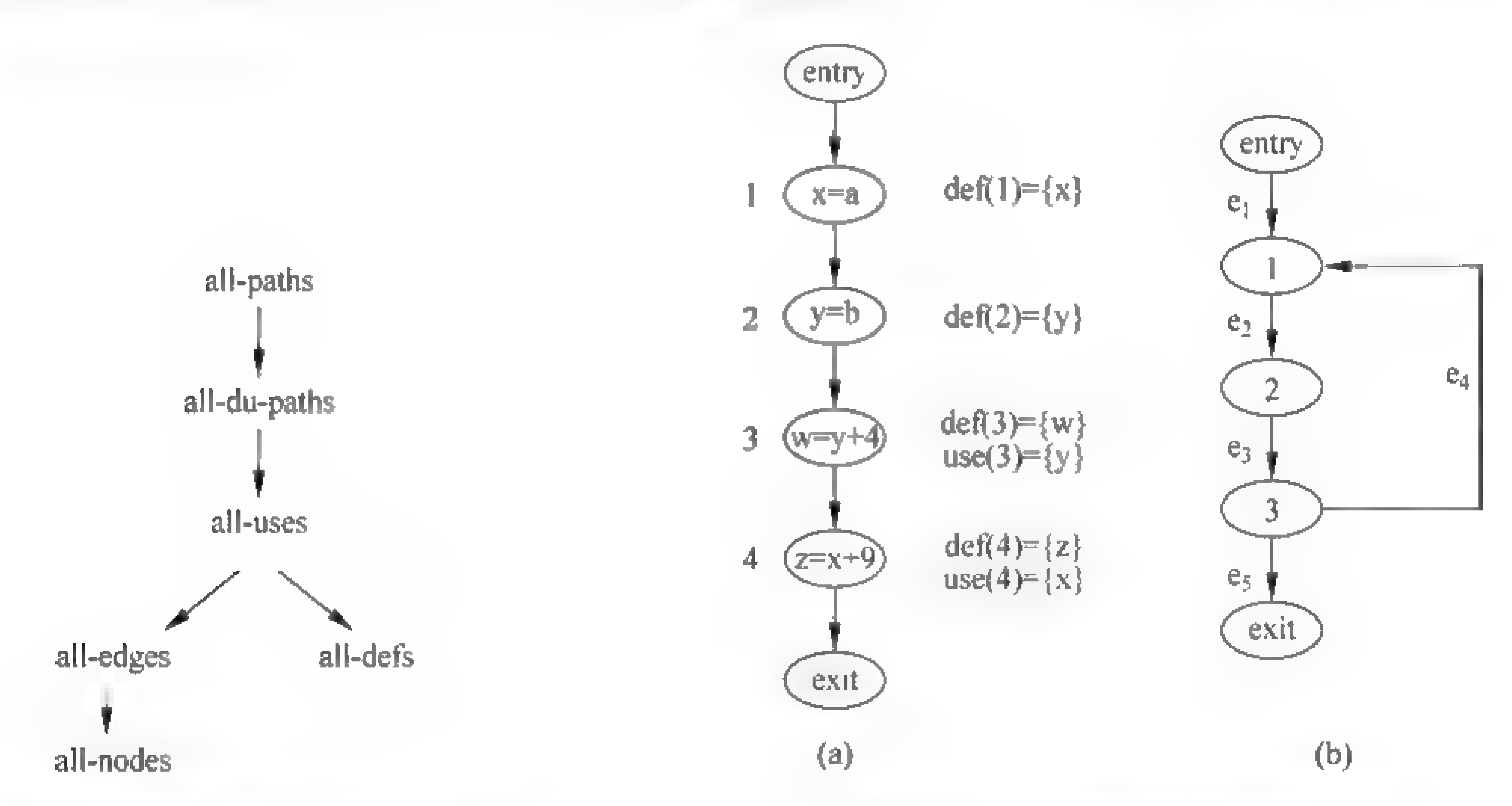


图 2-11　数据流测试标准涵盖关系　　**图 2-12　数据流测试标准涵盖关系**

2.2.3　循环测试

循环是大多数软件实现算法的重要部分，但是在软件测试时却很少注意它们。循环测试是一种白盒测试技术，注重于循环构造的有效性。有 4 种循环[3]：简单循环、串接循环、嵌套循环和不规则循环(如图 2-13 所示)。

1. 简单循环

下列测试集可以用于简单循环，其中 n 是允许通过循环的最大次数。

(1) 跳过整个循环。

(2) 只执行一次循环。

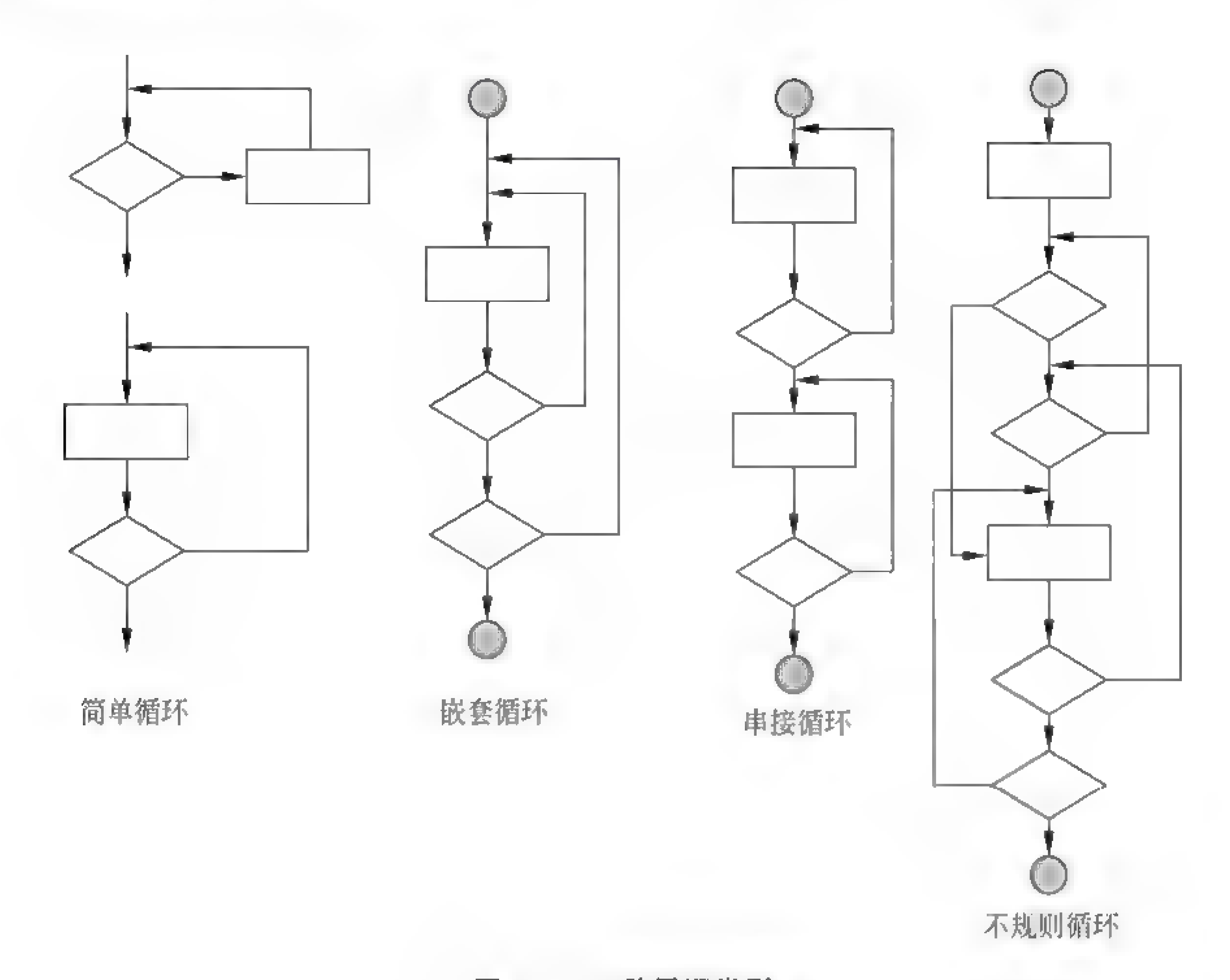


图 2-13　4 种循环类型

(3) 执行两次循环。

(4) 执行 m 次循环，其中 $m<n$。

(5) 执行 $n-1$、n、$n+1$ 次循环。

2. 嵌套循环

如果要将简单循环的测试方法用于嵌套循环，那么可能的测试数就会随嵌套层数成几何级增加，这会导致不实际的测试数目，Beizer 提出了一种减少测试数的方法：

(1) 从最内层循环开始，将其他循环设置为最小值。

(2) 对最内层循环使用简单循环测试，而使外层循环的迭代参数(即循环计数)最小，并增加其他的测试用例来测试范围外或排除的值进行测试。

(3) 由内向外构造下一个循环的测试，但其他的外层循环为最小值，并使其他的嵌套循环为“典型”值。

(4) 继续直到测试完所有的循环。

3. 串接循环

如果串接循环的循环都彼此独立，则可以使用简单循环测试策略来测试串接循环。但

是，如果两个循环串接起来，而第一个循环的循环计数是第二个循环的初始值，则这两个循环并不是独立的。如果循环不独立，则推荐使用嵌套循环的方法进行测试。

4. 不规则循环

尽可能将这类循环重新设计为结构化的程序结构。

白盒测试动态方法是使用程序设计的控制结构导出测试用例。使用这类方法，软件工程师能够产生测试用例：

(1) 保证一个模块中的所有独立路径至少被使用一次。

(2) 对所有逻辑值均需测试 true 和 false。

(3) 在上下边界及可操作范围内运行所有循环。

(4) 检查内部数据结构以确保其有效性。和动态方法比，白盒测试静态方法简单易学且使用成本低，一般在进行前者之前先应用静态方法。下面介绍 3 种代码检查法：代码审查、桌面检查和走查。

2.3 代码检查法

代码检查法主要检查代码和设计的一致性，代码对标准的遵循、可读性，代码逻辑表达的正确性，代码结构的合理性等方面；发现违背程序编写标准的问题，程序中不安全、不明确和模糊的部分，找出程序中违背编程风格的问题，包括变量检查、命名和类型检查、程序逻辑检查、程序语法和结构检查等内容。

2.3.1 代码审查

在 20 世纪 70 年代中期，Michael Fagan 在 IBM 制定出了审查的过程(Fagan 1976)[13]，其他人在此基础上又做了扩展和修改(Gilb and Graham 1993)[14]。该过程被认为是软件业最佳的实践(Brown 1996)[15]。人们可以审查任何一种软件工作产品，包括需求和设计文档、源代码、测试文档及项目计划等。审查定义为多阶段过程，涉及由受过培训的参与者组成的小组，他们把重点放在查找工作产品缺陷上。审查提供了一个质量关卡，文档在最终确定以前，必须通过该关卡的检查。虽然，对于 Fagan 的方法是否最有效并且是不是最有效的审查的形式还存在争议(Glass，1999)[16]，但是审查是强有力的质量技术，这是毫无疑问的。

1. 代码审查小组

代码审查是由若干程序员和测试人员组成一个审查小组，通过阅读、讨论和争议，对程

序进行静态分析的过程。审查小组通常由4类角色组成：主持人、作者、评论员和记录员。审查的一个关键特征就是每个人都要扮演某一个明确的角色。下面介绍各类角色。

1）主持人

主持人负责保证审查以既定的速度进行，使其既能保证效率，又能发现尽可能多的错误。主持人在技术上面必然能够胜任——虽然不一定是被检查的代码方面的专家，但必须能够理解有关的细节。主持人还负责管理审查的其他方面，例如分派审查代码的任务、分发审查所需的核对表、预定会议室、报告审查结果以及负责跟踪审查会议上指派的任务。

2）作者

作者是直接参与代码设计和编写的人，该角色在审查中扮演相对次要的角色。审查的目标之一就是让代码本身能够表达自己。如果它不够清晰，那么就需要向作者分配任务，使其更加清晰。除此之外，作者的责任就是解释代码中不清晰的部分，偶尔还需要解释那些看起来好像有错的地方为什么实际是可以接受的。如果参与评论的人对项目不熟悉，作者可能还需要陈述项目的概况，为审查会议做准备。

3）评论员

评论员是同代码有直接关系，但又不是作者的人。测试人员或者高层架构师也可以参与。评论员的责任是找出缺陷，他们通常在为审查会议做准备的阶段就已经找出了部分缺陷，然后随着审查会议中对代码的讨论，他们应该能够找出更多的缺陷。

4）记录员

记录员将审查会议期间发现的错误，以及指派的任务记录下来。作者和主持人都不应该担任记录员。

一般来说，参与审查的人数不应该少于3人，少于3个人就不可能有单独的主持人、作者和评论员了，因为这3种角色不应该被合并。传统的建议是限制参与审查的人数在6人左右，因为如果人数过多，那么这个小组就变得难以管理。

2. 代码审查的步骤

图2-14详细描述代码审查的一般步骤。

1）计划(Plan)

作者将代码提交给主持人。主持人决定哪些人复查这些材料，并决定会议在什么时间什么地点召开。接下来主持人会将代码，以及一个要求与会者注意的核对表分发给各人。材料应该打印出来，并且每行应当有行号，以便在会议中更快标识出错误的位置。

2）概述(Overview Meeting)

当评论员不熟悉他们要审查的项目时，作者可以花大约一个小时来描述一下这些代码的技术背景。加入概述也许有风险，因为这往往导致被检查的代码中不清晰的地方被掩饰。代码本身应该可以自我表达，在概述中不应该谈论它们。

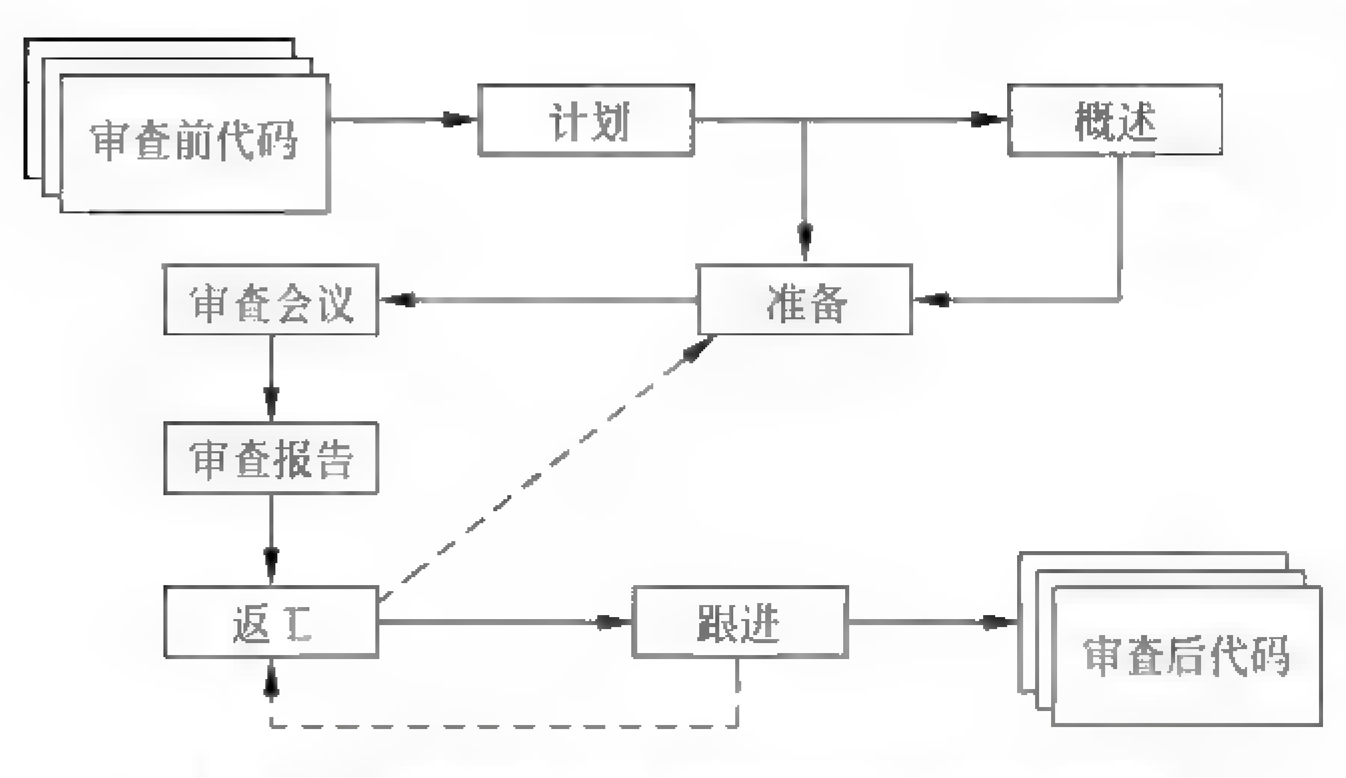


图 2-14 代码审查步骤(虚线框表示可重复步骤)

3) 准备(Preparation)

每一个评论员独立地对代码进行审查,找出其中的错误。评论员使用核对表来指导他们对材料的审查。

4) 审查会议(Inspection Meeting)

主持人挑选出除作者之外的某个人来阅读代码。所有的逻辑都应当有解释,包括每个逻辑结构的每个分支。在此过程中,其他小组成员可以提出问题,展开讨论,审查错误是否存在。实践证明,作者在讲解过程中能发现许多原来自己没有发现的错误,而讨论则促进了问题的暴露。在陈述期间,记录员需要记录发现的错误,但是所有的讨论应当在确认这是一个错误的时候停止。当记录员将错误的类型和严重程度记录下来以后,审查工作继续向下进行。如果一直在对某个问题不停的争论,那么主持人就应当敲桌子(摇铃)引起大家的注意,以使讨论回到正轨。

对代码的思考速度不能够太慢或者太快。如果速度太慢,那么大家的注意力就会不集中,这样的会议是不会富有成效的;如果速度太快,那么小组可能会忽视某些本应该被发现的问题。一个理想的审查速度应该随着环境的不同有很大变化。应保留以前的记录,这样以后就可以逐渐知道你所在的环境的最佳速度是怎样的。

不要在开会的过程中讨论解决方案,小组应该把注意力保持在识别缺陷上。某些审查小组甚至不允许讨论某个缺陷是否确实是一个缺陷。他们认为如果某个人对某个问题有困惑,那么就应该认为是一个缺陷了,设计、代码或者文档应该进一步清理。会议期间要避免外部干扰。通常会议不应该超过两个小时,最佳在90～120分钟之间。因为这样的会议是很耗费脑力的,过长的会议会导致效率低下。大多数的审查每小时讨论150行左右的代码。因此,较大规模的程序最好分多次审查,每一次处理一个模块或子程序。同理,一天安排超过一个审查会议也是不明智的。

5) 审查报告(Report)

一天的审查会议之后,主持人要写出一份审查报告(以E mail或其他类似形式),列出

每一个缺陷，包括它的类型和严重级别。审查报告有助于确保所有的缺陷都得到修正，它还可以用来开发一份核对表，强调与该组织相关的特定问题。

6）返工(Rework)

主持人将缺陷分配给某人来修复，这个人通常是作者。得到任务的人负责修正列表中的每个缺陷。

7）跟进(Follow-up)

主持人负责监督在审查过程中分配的返工任务。根据发现错误的数量和这些错误的严重级别，跟踪工作进展的方式可以是让评论员重新审查整个工作成果，或者让评论员只重新审查修复的部分，或者允许作者只完成修改而不做任何跟进。

有的书提到在审查会议之后第三小时的会议。虽然在进行审查的期间，与会者不允许讨论所发现问题的解决方案，但还是可能有人想对此进行讨论。你可以主持一个非正式的第三个小时的会议，允许有兴趣的人在正式审查结束之后讨论解决方案。

为了使审查过程更富有效果，需要树立起对审查的正确态度。如果代码的作者认为审查是对自己人格的攻击并采取一种防御的态度，那么审查过程将会没什么效率。相反，作者必须采取一种积极和建设性的态度：审查的目的是为了找出程序中的错误，进而改进程序的质量。因此，大多数人建议审查的结果不公开，只有与会者知道。类似地，审查的结果不应该作为员工表现的评估标准。在审查中被检验的代码仍处于开发阶段，对员工的评估应当基于最终产品，而不是尚未完成的工作。所以，在审查的时候让经理参与通常不是一个好主意。软件审查的要点是，这是一个纯技术性的复查。经理的出席会对交流产生影响：人们会觉得他们不是在审查各种材料，而是在被评估，关注的焦点就会从技术问题转换到行政问题上了。不过经理有权知道审查的结果，应当准备一份审查报告让经理了解情况。

审查过程除了能够发现代码中的缺陷这一主要作用，还产生其他一些有益的效果。

(1) 作者通常会获得关于编程风格、算法选择和编程技巧方面的反馈。其他与会者也会从暴露出的问题中获得经验。

(2) 审查过程在早期就确定了程序中容易出错的部分，有助于在以后的自动测试过程中对这些部分重点关注。

在审查过程中一个重要的部分是使用一个核对表检查程序中的常见错误。但是许多核对表过多地关注程序风格的问题而不是错误，比如“注释是否准确和有意义”“if else、do-while 等代码块是否对齐”这样的项目。还有一些核对项表述太模糊而不实用，比如“代码是否满足设计要求”这样的表述。表 2-1 给出的核对表是从多年实践中总结出来的程序中的常见错误，它把程序中可能发生的各种错误进行分类，对每一类型列举出尽可能多的典型错误。核对表基本上是同编程语言无关的，其中大多数错误都可能在任何语言中出现。也可以根据使用的编程语言和自己的编程实践补充这份核对表。

表 2-1　核对表的示例

数据引用错误
1. 引用的变量是否未初始化? 2. 数组的下标是否越界? 3. 数组的下标是否是整数值? 4. 指针或引用变量指向的内存是否已被分配? 5. 不同的数据类型指向同一内存区域时,当通过某一类型变量引用时,内存中的值是否和该变量为同一类型? 6. 当分配的内存空间比可寻址的内存单元小时是否有明显或不明显的寻址问题? 7. 使用指针或引用变量时,是否与引用的值具有相同的类型? 8. 当一个数据结构在多个子程序中使用时,该数据结构是否在每个子程序中定义一致? 9. 对字符串进行读写操作或引用数组下标时,是否有超出了字符串或数组末尾而导致的 off-by-one 错误? 10. 对于面向对象的语言,所有的继承条件是否都在实现类中满足?
数据声明错误
1. 是否所有的变量都显式地声明? 2. 变量在声明时初始化是否合适? 3. 每个变量是否有正确的数据类型? 4. 变量的初始化是否和数据类型一致? 5. 不同的变量是否有相似的名字?(*)
计算错误
1. 是否有不同类型的数据混合在一起计算? 2. 计算表达式过程中是否会发生上溢或下溢? 3. 除法操作的除数是否为 0? 4. 是否会有不准确的计算结果? 5. 变量值是否超出了实际应用的取值范围? 6. 在由多个操作符构成的表达式中,对于计算的顺序和操作符优先级的假设是否正确? 7. 是否有对整数算术的非法使用,特别是除法?
比较错误
1. 是否有不同类型之间的比较? 2. 比较操作符是否正确? 3. 每一个布尔表达式是否得到预期的结果? 4. 布尔操作符的操作数是否为布尔类型? 5. 是否有小数和浮点数之间的比较? 6. 在由多个布尔操作符构成的表达式中,计算的优先级是否如预期? 7. 编译器计算布尔表达式的方式是否影响程序?

续表

控制流错误
1. 每个循环是否能中止? 2. 每个程序、模块或子程序是否会中止? 3. 是否有可能因为某些取值而使某些循环从来不会被执行? 如果有,是否是疏忽? 4. 控制条件中是否有 off-by-one 错误? 5. 对每个开括号是否有对应的闭括号? 6. 是否有没考虑到的情况?
接口错误
1. 子程序接受的参数数目是否和调用程序传递的参数数目相同? 顺序是否正确? 2. 每个参数的类型是否和声明的类型一致? 3. 每个参数使用的单位系统是否和声明时的一致? 4. 调用内置函数时,参数的数目、类型和顺序是否正确? 5. 子程序是否改变了只作为输入参数传入的参数值? 6. 如果使用了全局变量,那么在所有引用它们子程序中是否具有一致的定义? 7. 常数是否作为参数传入?
输入/输出错误
1. 文件是否被显式地声明? 它们的属性是否正确? 2. 在文件的打开语句中属性是否正确? 3. 文件的格式规范是否和 I/O 语句中的信息一致? 4. 是否有足够的内存保存读入的文件数据? 5. 所有的文件是否在使用前都打开? 6. 所有的文件是否在使用后都关闭? 7. 文件结束的情况是否被考虑并正确处理? 8. I/O 错误是否被正确处理? 9. 在程序打印出或显示的文本中是否有拼写和语法错误?

总结代码审查方法,可以概括出以下 9 个方面的特点:

- 审查专注于缺陷的检测,而非修正。
- 审查人员要为审查会议做好预先准备,并且带来一份他们发现的已知问题列表。
- 参与者都被赋予明确的角色。
- 审查的主持人不是被检查产品的作者。
- 审查的主持人应该已经接受过主持审查会议方面的培训。
- 只有在与会者都做好准备之后才会召开审查会议。
- 每次审查所收集的数据都会被应用到以后的审查中,以便对审查进行改进。
- 高层管理人员不参加审查会议。

• 核对表关注的是审查者过去所遇到的问题。

2.3.2　桌面检查

桌面检查是一种人工检查程序的方法，通过对源程序代码进行分析、检验来发现程序中的错误。桌面检查关注的是变量的值和程序逻辑，所以执行桌面检查要严格按照程序中的逻辑顺序。检查人员使用笔和纸记录下检查结果。

比较正式的桌面检查可以使用表格的形式来记录检查的结果，表格的设计如下：

(1) 第一列是行号(源程序中可能没有行号，但在桌面检查中标明正在检查的行是很有必要的，这可以使检查过程清晰明了)。

(2) 待检查程序中使用的每个变量占据一列。变量名作为列标题，最好按照字母顺序排列。随着程序流程的执行，新的变量值填入对应的表格中。如果变量名由多个单词构成，可以用空格把多个单词分开。比如对于变量名 discountPrice，可以使用 discount Price 作为列标题。

(3) 条件(Conditions)列。条件的结果可能为真(T)或假(F)。随着程序的执行，条件值被计算出来并记录到表格中。这可以用在任何需要计算条件值的地方——if、while 和 for 语句都有计算条件值的含义。

(4) 输入输出(Input/Output)列。这列用来记录需要用户输入的值和程序输出的值。输入数据可以表示为：变量名+"?"+变量值，例如 price?200；输出数据可以表示为：变量名+"="+变量值，例如 discount=180。

下面通过几个例子来说明如何使用桌面检查技术。

例 2.1　包含顺序执行语句。

问题描述：计算商品打折后的价格(注：正式的桌面检查中不必注明详细的问题描述，这里只是为了帮助理解伪代码)。

伪代码：

```
calcDiscountPrice()
    Input price,discountPercent
    discount=price * discountPercent/100
    discountPrice=price-discount
    Display discountPrice
STOP
```

测试数据：

输入：price=$200，discountPercent=10(表示 10%)。

正确结果：discount=$20，discountPrice=$180。

根据前面所建议的表格格式，第一列 LN 表示代码行号，最后一列 Input/Output 表示输入输出数据。中间各列为代码中出现的变量名(按字母顺序排列)，把由两个字母构成的

变量名用空格分隔开，如表2-2所示。

表 2-2　例 2.1 的结果

LN	discount	discount Percent	discount Price	price	Input/Output
1					
2		10		200	price?200； discountPercent?10
3	200 * 10/100=20				
4			200−20=180		
5					discountPrice=180
6					

例 2.2　包含选择语句 if-else。

问题描述：根据用水量和用户类型(家用或商用)计算水费。

伪代码：

```
calcWaterCost()
  Input waterUsed
  IF waterUsed < 100 THEN
      cost=50
  ELSE
      Input customerType
      IF customerType="D" THEN
          cost=waterUsed * 0.5
      ELSE
          cost=waterUsed * 0.6
          Display "Commercial rate"
      ENDIF
      Display "High usage"
  ENDIF
  Display cost
STOP
```

测试数据：

输入：waterUsed=200，customerType=D。

正确结果：cost=100。

第一列 LN 表示代码行号，最后一列 Input/Output 表示输入输出数据，倒数第二列 Conditions 记录代码中出现的各个条件表达式的值。中间各列为代码中出现的变量名(按字母顺序排列)，把由两个字母构成的变量名用空格分隔开，如表2-3所示。

表 2-3 例 2.2 的结果

LN	cost	customer Type	water Used	Conditions	Input/ Output
1					
2			200		waterUsed?200
3				200<100?is F	
4					
5					
6		D			customerType?D
7				D=D?is T	
8	200 * 0.5=100				
9					
10					
11					
12					
13					High usage
14					
15					cost=100
16					

例 2.3 包含循环语句 for。

问题描述：计算 x 的平方，x 从 1～3。

伪代码：

```
1 calcSquares()
2   Display "X","X Squared"
3   FOR x=1 TO 3 DO
4       xSquared=x * x
5       Display x,xSquared
6   ENDFOR
7   Display "-----------"
8 STOP
```

测试数据：

输入：无。

正确结果：x=1,xSquared=1；x=2,xSquared=4；x=3,xSquared=9。

第一列 LN 表示代码行号，最后一列 Input/Output 表示输入输出数据，倒数第二列 Conditions 记录代码中出现的各个条件表达式的值。中间各列为代码中出现的变量名(按字母顺序排列)，如表 2-4 所示。

表 2-4 例 2.3 的结果

LN	x	xSquared	Conditions	Input/ Output
1				
2				x,xSquared
3	1		1<=3?is T	
4		1*1=1		
5				x=1,xSquared=1
6	1+1=2			
3			2<=3?is T	
4		2*2=4		
5				x=2,xSquared=4
6	2+1=3			
3			3<=3?is T	
4		3*3=9		
5				x=3,xSquared=9
6	3+1=4			
7				-----------
8				

例 2.4 包含子程序。

问题描述：查找 3 个数中的最小值。

伪代码：

```
1  compareNumbers()
2    Input number1,number2,number3
3    getSmallestNumber()
4    Display smallest
5  STOP
6  getSmallestNumber()
7    smallest=number1
8    IF number2 < smallest THEN
9       smallest =number2
10   ENDIF
11   IF number3 < smallest THEN
12      smallest=number3
13   ENDIF
14 EXIT
```

测试数据：

输入：number1=5，number2=3，number3=8。

正确结果：smallest=3。

第一列 LN 表示代码行号，最后一列 Input/Output 表示输入输出数据，倒数第二列

Conditions 记录代码中出现的各个条件表达式的值。中间各列为代码中出现的变量名(按字母顺序排列),如表 2 5 所示。

表 2-5 例 2.4 的结果

LN	number1	number2	number3	smallest	Conditions	Input/ Output
1						
2	5	3	8			number1?5; number2?3; number3?8
3						
6						
7				5		
8					3<5?is T	
9				3		
10						
11					8<3?is F	
12						
13						
14						
4						smallest=3
5						

桌面检查可以由程序作者本人来执行,但这对于大多数程序员来说效率并不高。因为这违反了一条测试原则 —— 人们在测试自己的程序时效率通常是很低的。所以,桌面检查最好由另 一个人来执行而不是程序作者本人(比如可以两个程序员互相检查对方的程序)。但这种方法不如审查或走查过程有效,因为审查或走查需要 一个团队,团队会营造 一种健康的竞争环境,团队成员以找出错误来体现自己的价值。而桌面检查过程只有 一个人在阅读代码,没有团队成员之间的协作效应。总之,桌面检查比什么都不做要好,但不如审查或走查有效。

2.3.3 走查

代码走查和代码审查具有很多相同的步骤,但在查找错误的方法上有些小小的不同。和审查 一样,走查也是 一次持续 一到两小时的会议。走查团队由 3~5 人组成。其中 一人扮演审查中的主持人角色, 一人扮演审查中的记录员角色, 一人扮演测试者的角色。当然代码作者也是其中之 一。其他的与会者可以包括: 一个很有经验的程序员、编程语言的专家、初级程序员(新手,可以从新鲜的无偏见的视角看问题)、最终维护程序的人、其他项目组的成

员、同一编程小组的成员。

开始的步骤和代码审查过程一样：提前几天把相关材料分发给与会者，让他们认真研究程序，然后开会。会议的进程与代码审查不同，不是简单地读程序和对照核对表进行检查，而是让与会者"充当"计算机。首先由测试者为所测程序准备一批有代表性的测试用例，提交给走查小组。在会议上，与会者集体扮演计算机的角色，让测试用例沿程序的逻辑运行一遍，随时记录程序的踪迹和状态(也就是各变量的值)，供分析和讨论用。

当然用例数量不能太多、太复杂，因为人们"执行"程序的速度要比计算机慢很多。用例本身并不起主要作用，它们只是作为媒介来向代码作者提出有关程序设计和逻辑方面的问题。在大多数走查过程中，更多的错误是在提问的过程中，而不是直接运行测试用例的过程中被发现。

和审查过程一样，与会者的态度是关键。评论只针对程序，而不针对程序员。换句话说，出现错误不要看作是代码作者的问题，而是软件开发过程中固有的难点。走查也有和审查一样的后续步骤，审查所带来的好处同样适用于走查。

2.4　总结

白盒测试是一种基于源程序或代码的测试方法，分为静态和动态两种类型。静态方法是指按一定步骤直接检查源代码来发现错误，而不用生成测试用例并驱动被测程序运行来发现错误，也称为代码检查法；动态方法是指按一定步骤生成测试用例并驱动被测程序运行来发现错误。静态方法有桌面检查、代码审查及走查，动态方法有基本路径测试、条件测试、数据流测试及循环测试。

白盒测试是一种主要的单元测试方法，一般由软件开发人员进行。白盒测试过程主要有 5 个步骤：根据源程序画程序图、生成测试用例、执行测试、分析覆盖标准、判定测试结果。

2.5　参考文献

[1]　McCabe, T. (1976). A Software Complexity Measure. IEEE Transactions on Software Engineering SE-2: pp. 308～320

[2]　R. Pressman. Software Engineering: A Practitioner's Approach. Boston: McGraw Hill, 2005

[3]　Boris Beizer. Software Testing Techniques. International Thomson Computer Press. 2nd edition. June 1990

[4] Glenford J. Myers. The Art of Software Testing. John Wiley & Sons. 1 edition. February 20, 1979
[5] White, L. J, E. I cohen. A Domaion Strategy for Program Testing. IEEE Trans. Software Engineering. vol. SE 6, no. 5, May 1980, pp. 247～257
[6] Howden, W. E.. Weak Mutation Testing and the Completeness of Test Cases. IEEE Trans. Software Engineering. vol. SE 8, no. 4, July 1982, pp. 371～379
[7] Foster, K. A.. Sensitive Test Data for Boolean Expressions. ACM Software Engineering Notes. vol. 9, no. 2, April 1984, pp. 120～125
[8] Tai, K. C. and H. K. Su. Test Generation for Boolean Expressions. Proc. COMPSAC'87. October 1987. pp. 278～283
[9] Tai, K. C. What to Do Beyond Branch Testing. ACM Software Engineering Notes. vol. 14, no 2, April 1989, pp. 58～61
[10] Frankl, P. G., E. J. Weyuker. An Application Family of Data Flow Testing Criteria. IEEE Trans. Software Engineering. vol. SE-14, no. 10, October 1988, pp. 1483～1498
[11] Ntafos, S. C. A Comparison of Some Structural Testing Strategies. IEEE Trans Software Engineering. vol. SE-14, no. 6, June 1988, pp. 868～874
[12] Frankl, P. G., S. Weiss. An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow. IEEE Trans. Software Engineering. vol. SE-19, no. 8, August 1993, pp. 770～787
[13] Michael E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal. 15(3), 1976, pp. 182～211
[14] Tom Gilb, Dorothy Graham. Software Inspection. Wokingham, England: Addison-Wesley, 1993
[15] Norm Brown. Industrial-Strength Management Strategies. IEEE Software. 13(4), 1996, pp. 94～103
[16] Robert L. Glass. Inspections-Some Surprising Findings. Communications of the ACM 42(4), 1999, pp. 17～19

2.6 思考与练习

1. 描述什么是独立路径集。描述基于基本路径测试的测试过程。

2. “新增的独立路径至少有一个边尚未被访问过”，该原则是否正确，为什么？

3. 条件测试可以测试什么样的错误？为什么？

4. 数据流的测试方法为什么要求“定义清纯(Definition-clear)”？

5. 回归测试的两种模式是什么？是否还有其他模式(举例说明)？

6. 什么是波及效应分析？它的适用范围是什么？为什么说波及效应分析是一个迭代过程？

7. 什么是向后的程序切片？什么是向前的程序切片？作为切片标准里的语句一定会在切片结果集里吗？为什么？

8. 回归测试有哪些消耗？

2.7 进一步阅读

IEEE (1987). ANSI/IEEE Standard pp. 1008 1987. IEEE Standard for Software Unit Testing.

IEEE (1990). IEEE Standard 610. pp. 12～1990. IEEE Standard Glossary of Software Engineering Terminology

Kaner, C., J. Falk, et al. (1999). Testing Computer Software. New York, Wiley Computer Publishing

Paul C. Jorgensen, Software Testing. A Craftsman's Approach. Second Edition, CRC Press, 2002

第3章

黑 盒 测 试

黑盒测试注重测试软件的功能性需求，即黑盒测试需要软件工程师生成输入条件集来检测程序所有功能需求。黑盒测试并不是白盒测试的替代品，而是配合白盒测试发现其他类型的错误。黑盒测试试图发现以下类型的错误：功能不对或遗漏、界面错误、数据结构或外部数据库访问错误、性能错误以及初始化和终止错误。

白盒测试(参见第2章)在测试的早期执行，而黑盒测试主要用于测试的后期。黑盒测试故意不考虑控制结构，而是关注信息域。黑盒测试用于回答以下问题：

- 如何测试功能的有效性？
- 何种类型的输入会产生好的测试用例？
- 系统是否对特定的输入值尤其敏感？
- 如何分隔数据类的边界？
- 系统能够承受何种数据传输率和数据量？
- 特定类型的数据组合会对系统产生何种影响？

运用黑盒测试，可以导出满足以下标准的测试用例集[1]：

(1) 所设计的测试用例能够减少达到合理测试所需的附加测试用例数。

(2) 所设计的测试用例能够告知某些类型错误的存在或不存在，而不是仅仅与特定测试相关的错误是否存在。

快速阅览：

什么是黑盒测试？黑盒测试又叫做功能测试，是基于系统已实现的功能进行测试的。使用该方法的具体的测试用例设计方法包括等价类划分法、边界值分析法、正交数组测试法、因果分析法等。

由谁来负责黑盒测试？黑盒测试一般由有经验的软件测试人员完成集成测试、功能测试的计划和执行。

为什么黑盒测试如此重要？黑盒测试试图发现以下类型的错误：功能不对或遗漏、界

面错误、数据结构或外部数据库访问错误、性能错误以及初始化和终止错误。

黑盒测试步骤各是什么？ 黑盒测试过程主要包括如下4个步骤：根据软件规格说明书生成测试用例，执行测试，分析覆盖标准，判定测试结果。

有哪些工件形成？ 在一些情况下，会生成功能测试计划、系统模型和测试用例。黑盒测试结果存档以便将来软件维护时使用。

如何确保我们准确地完成了任务？ 尽管永远不能保证已经执行了所有可能的黑盒，但能肯定测试已经发现了错误(并且已修正了这些错误)。另外，如果已经制定了一个白盒测试计划，则可以检查以保证所有计划测试已被完成。

3.1 基于图的测试方法

黑盒测试的第一步是理解软件所表示的对象及其关系，然后，第二步是定义一组保证“所有对象与其他对象具有所期望的关系”[2]的测试序列，换言之，软件测试首先是创建对象及其关系图，然后导出测试序列以检查对象及其关系并发现错误。

为了完成这些步骤，软件工程师首先创建一个图，节点代表对象，连接代表对象间的关系，节点权值描述节点的属性(如特定的数据值或状态行为)，连接权值描述连接的特点。

图的符号表示如图3-1(a)所示。节点表示为圆，而连接有几种，有向连接(有箭头表示)表明关系只在一个方向上存在。双向连接，也称为对称连接，表示关系适于两个方向。如果节点间有几种联系，就使用并行边。

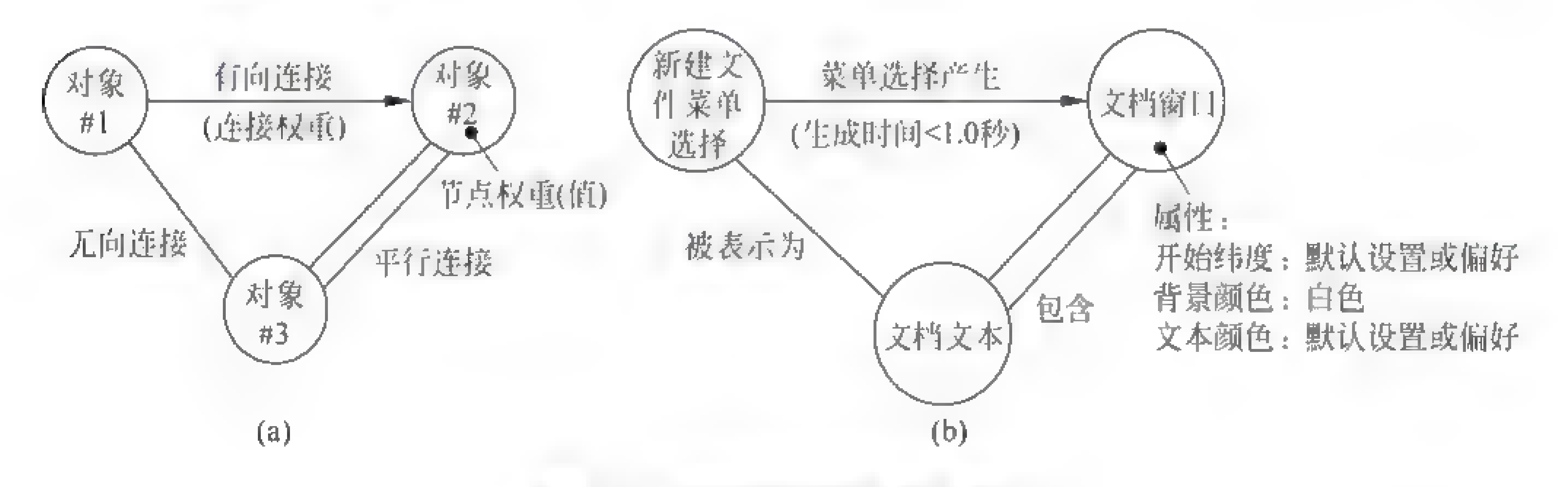


图3-1 图的符号表示及简单例子

举一个简单的例子，考虑字处理应用的部分程序图，如图3-1(b)所示。

对象#1=新建文件菜单选择
对象#2=文档窗口
对象#3=文档文本

如图3-1所示，选择菜单“新建文件”产生一个文档窗口，文档窗口的节点权值提供窗口产生时所期望的属性集，连接权值表明窗口必须在1.0s之内产生，一条无向边在“选择菜单

新建文件"和"文档文本"之间建立对称联系,并行连接表明"文档窗口"和"文档文本"间的联系,事实上,要产生测试用例还需要更加详细的图。软件工程师遍历图,并覆盖所显示的联系就可以导出测试用例,这些用例用于发现联系之间的错误。Beizer 描述了几个使用图的行为测试方法:

(1) **事务流建模**。节点代表事务的步数(如使用联机服务预订航空机票的步数),连接代表步骤之间的连接关系(如 flighgt. information. input 后跟 validation/availabilty. processing)。数据流图可用于辅助产生这种图。

(2) **有限状态建模**。节点代表用户可观测的不同软件状态(如订票人员处理订票时的各个屏幕),而连接代表状态之间的转换(如在 inventory-availability-look-up 时验证 order-information 并后跟 customer-billing-information-input)。状态变迁图可用于辅助产生这种图。

(3) **数据流建模**。节点是数据对象,而连接是将数据对象转换为其他对象时发生的变换。例如,节点 FICA. tax. withheld(FTW)由 gross. wagess(GW)利用关系 FTW=0.062×GW 计算而来。

(4) **时间建模**。节点是程序对象,而连接是对象间的顺序连接。连接权值用于指定程序执行时所需的执行时间。

基于图的测试方法的详细讨论超出了本书的范围,感兴趣的读者可以阅读参考文献 Boris Beizer(1995)一书。但是,大致了解一下基于图的测试还是值得的。

基于图的测试开始定义节点和节点权值,也即标识对象及其属性,数据模型可以作为起始点,但是要注意很多节点是程序对象(不在数据模型中时显表示出来),为了标识图的起点和终点,可以定义入点和出点。标识节点以后,就可以建立连接及其权值,连接一般应当命名,但是当代表程序对象间控制流的连接时除外。

很多情况下,图模型可能有循环(如图的路径含有环),循环测试(参见 3.3.3 节)也可用于行为(黑盒)测试,图可用于标识需要测试的循环。分别研究每个关系,以导出测试用例。研究顺序关系的传递性可以发现关系在对象间传播的影响。举例说明,有 3 个对象 X、Y 和 Z。考虑如下关系:

计算 Y 需要 X
计算 Z 需要 Y

所以,X 和 Z 之间有传递性:

计算 Z 需要 X

基于这种传递性,测试 Z 的计算时要考虑 X 和 Y 的各种值。

关系(图连接)的对称性也是设计测试用例的重要考虑,如果关系是双向(对称)的,就要测试这种性质。很多应用程序的 UNDO 功能[2]实现了有限的对称性。也就是说,当一个动作完成后,UNDO 功能容许撤销该动作。应该彻底测试这个功能并标识所有的例外情况(即不能使用 UNDO 的地方)。最后,图的每个节点都应当有到自己的关系,本质上是"空操

作”循环。自反性也应当进行测试。

开始设计测试用例时，第一个目标是节点的覆盖度，这意味着测试不应当遗漏某个节点，而且节点的权值是正确的。接着，考虑连接的覆盖率，基于属性测试每个关系，例如，测试对称关系以表明它的确是双向的，测试传递关系以表明存在传递性，测试自反关系以表明存在空操作。指明连接权值时，要设计测试以展示权值是否有效，最后，加入循环测试（参见3.3.3节）。

3.2 等价划分

等价划分是一种黑盒测试方法，将程序的输入域划分为数据类，以便导出测试用例。理想的测试用例是该用例能独自发现一类错误（如字符数据的处理不正确）。等价划分试图定义一个测试用例以发现各类错误，从而减少必须开发的测试用例数。

等价划分的测试用例设计基于输入条件的等价类评估。使用前面章节介绍的概念，如果对象由具有对称性、传递性或自反性的关系连接，就存在等价类[2]。等价类表示输入条件的一组有效或无效的状态。典型地，输入条件通常是一个特定的数值、一个数值域、一组相关值或一个布尔条件。可按照如下指南定义等价类：

(1) 如果输入条件代表一个范围，则可以定义一个有效等价类和两个无效等价类。

(2) 如果输入条件需要特定的值，则可以定义一个有效等价类和两个无效等价类。

(3) 如果输入条件代表集合的某个元素，则可以定义一个有效等价类和一个无效等价类。

(4) 如果输入条件是布尔式，则可以定义一个有效等价类和一个无效等价类。

作为例子，考虑自动银行应用软件所管理的数据，用户可以用自己的微机拨号到银行，提供6位数的密码，并遵循一系列键盘命令以触发各种银行功能。银行应用程序的软件可以接受如下格式的数据：

- 区号——空或3位数字。
- 前缀——3位数字，但不是0和1开始。
- 后缀——4位数字。
- 密码——6位字母或数字。
- 命令——“检查”、“存款”、“付款”等。

与银行应用程序各种数据元素相关的输入条件可以表示为：

(1) 区号。

输入条件，布尔值——区号存在与否。

输入条件，范围——定义在200和999之间的数值，带有特殊例外值。

(2) 前缀。

输入条件，范围——指定的数值大于200。

输入条件，值——4 位数字长度。

(3) 密码。

输入条件，布尔值——密码存在与否。

输入条件，值——6 位字符串。

(4) 命令。

输入条件，集合——包含事先表明的命令。

利用上述导出等价类的指南，就可以为每个输入域的数据项生成并执行测试用例，测试用例的选择最好是每次执行等价类的尽可能多的属性。

3.3 边界值分析

由于某些未被完全知道的原因，输入域的边界比中间更加容易发生错误，为此，可用的边界值分析(Boundary Value Analysis，BVA)可作为一种测试技术。边界值分析选择一组测试用例检查边界值。

边界值分析是一种补充等价划分的测试用例设计技术。BVA 不是选择等价类的任意元素，而是选择等价类边界的测试用例，BVA 不仅注重于输入条件，而且也从输出域导出测试用例[1]。

BVA 的指南类似于等价划分：

(1) 如果输入条件代表以 a 和 b 为边界的范围，测试用例应当包含 a、b、略大于 a、b 和略小于 a，b 的值。

(2) 如果输入条件代表一组值，测试用例应当执行其中的最大值和最小值，还应当测试略大于最小值的值和略小于最大值的值。

(3) 上述指南(1)和(2)也适用于输出条件，例如，工程分析程序要求输出温度和压强的对照表，测试用例应当能够创建包含最大值和最小值的项。

(4) 如果程序数据结构有预定义的边界(如数组有 100 项)，要测试其边界的数据项。

大多数软件工程师会在某种程度上自发地执行 BVA，利用上述指南，边界测试会更加完整，从而更有可能发现错误。

3.4 因果分析法

因果分析法(Cause-Effect Analysis)是一种测试用例的生成方法，它提供了对逻辑条件和对应的动作(Action)的一个简明的表示。因果分析法是一种黑盒测试方法，从分析软件

系统需求规格说明书开始，得到因果列表；基于此列表建立决策表，基于决策表的规则生成测试用例。图 3-2 用数据流图表示了因果分析法的过程。



图 3-2　因果分析法过程

因果分析法的过程的重要步骤是生成因果列表并在此基础上建立决策表。因果图法提供一种有效的因果关系描述方法，由因果图法转成决策表的方法也很直接。下面介绍因果图的基本图形符号和约束符号，并举例说明由因果图法转成决策表的方法。

3.4.1　因果图——图形符号

在因果图中通常用 C_i 表示原因，E_i 表示结果，原因与结果在图中用节点表示，当原因、结果出现时，称节点状态为 1，否则为 0。原因与结果之间的关系有以下 4 种（见图 3-3(a)～(d)）。

(1) 恒等(—)：若原因出现，则结果出现；若原因不出现，则结果也不出现（见图 3-3(a)）。

(2) 非(～)：若原因出现，则结果不出现；若原因不出现，则结果出现（见图 3-3(b)）。

(3) 或(∨)：若几个原因中有一个出现，则结果出现；若几个原因都不出现，则结果不出现（见图 3-3(c)）。

(4) 与(∧)：若几个原因都出现，则结果出现；若其中有一个原因不出现，则结果不出现（见图 3-3(d)）。

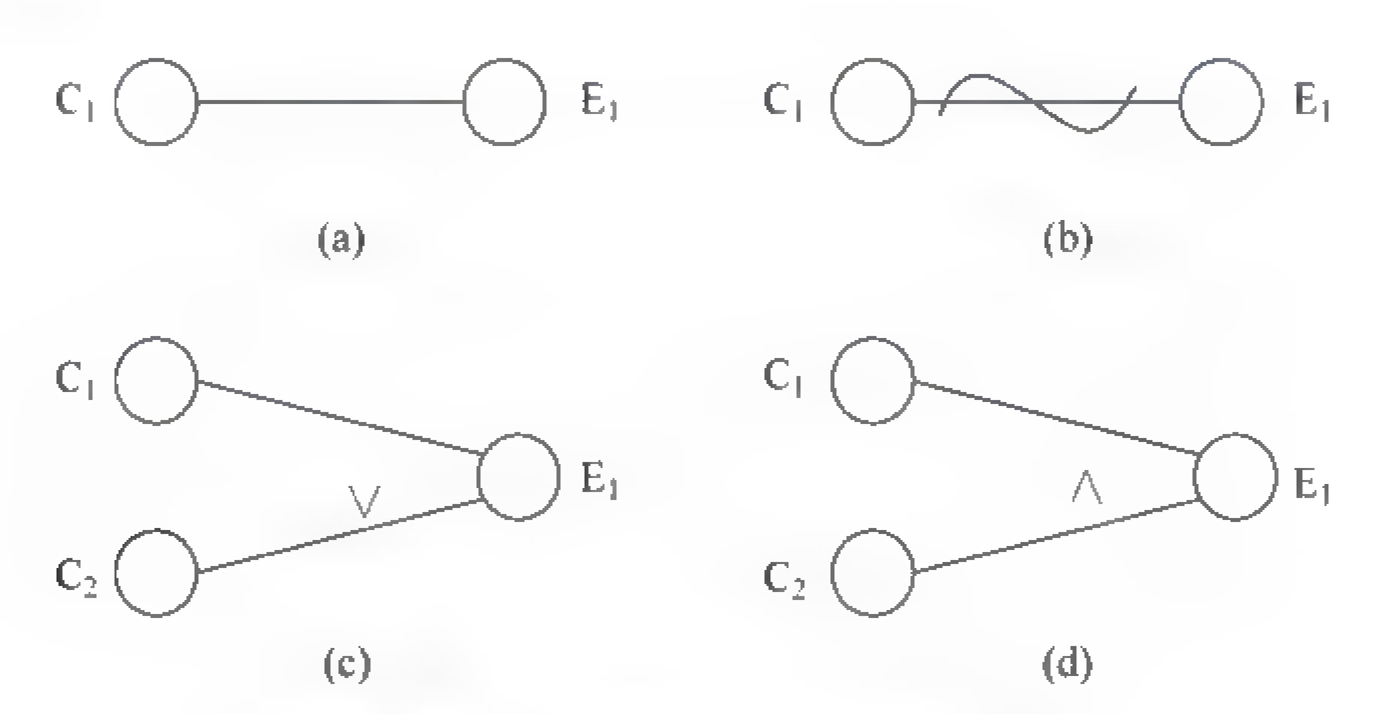


图 3-3　因果图的基本图形符号

为了表示原因与原因之间，结果与结果之间可能存在的约束条件，在因果图中可以附加一些表示约束条件的符号。从输入（原因）考虑，有 4 种约束：E（互斥）、I（包含）、O（唯一）和 R（要求）；从输出（结果）考虑还有一种约束 M（屏蔽）（见图 3-4(a)～(d)）。

从输入(原因)考虑：	
E(互斥)：	C_1、C_2 两个原因不能同时成立，两个中最多有一个能成立(见图 3-4(a))。
I(包含)：	C_1、C_2 和 C_3 3 个原因中至少有一个必须成立(见图 3-4(b))。
O(唯一)：	C_1 和 C_2 两个原因中必须有一个成立，且仅有一个成立(见图 3-4(c))。
R(要求)：	当 C_1 原因成立时，C_2 原因也必须成立，即不可能 C_1 成立而 C_2 不成立(见图 3 4(d))。
从输出(结果)考虑：	
M(屏蔽)：	当 E_1 是 1 时，E_2 必须是 0；而当 E_1 是 0 时，E_2 的值不定(见图 3-4(e))。

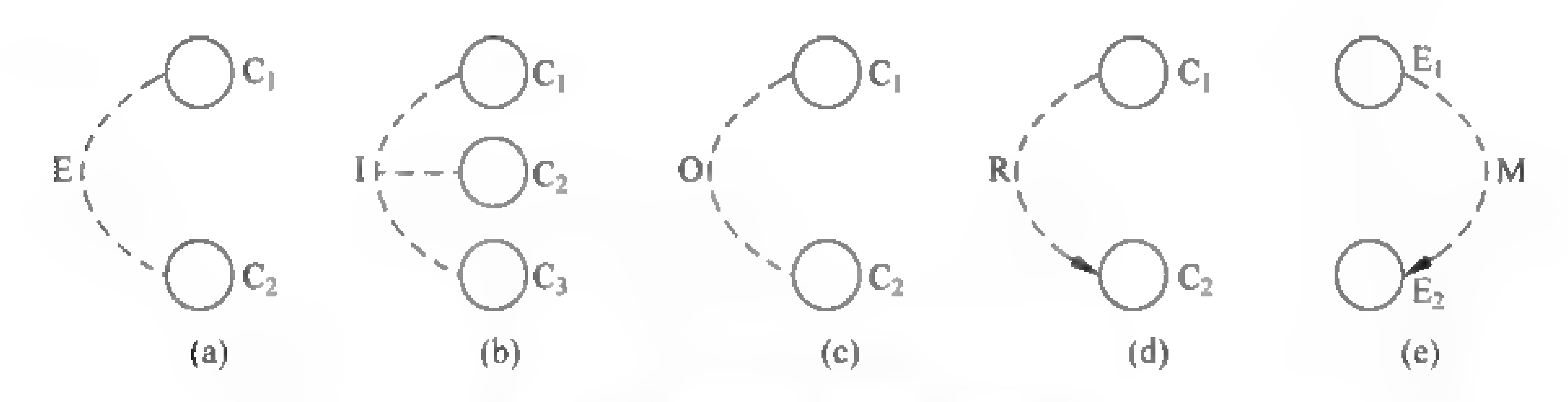


图 3-4　因果图的约束条件符号

3.4.2 因果图——举例

下面介绍一个小型因果图，假设有以下有关一个文件管理系统的一段规格说明：“在文件第一列的字符必须是一个 A 或 B，在文件第二列的字符必须是一个数字。在这种情况下，文件是被修改了。如果第一个字符不正确，则打印 X12 消息。如果第二个不是数字，则打印 X13 消息。”此段规格说明中的原因为：

C_1：第一列的字符是 A
C_2：第一列的字符是 B
C_3：第二列的字符是数字

此段规格说明中的结果为：

E_1：文件修改过
E_2：打印消息 X12
E_3：打印消息 X13

此段规格说明的因果图如图 3 5 所示。注意节点 C_4 是被创建的中间节点。C_1 和 C_2 都不成立时，E_2 成立；C_1 和 C_2 任一成立时，E_2 不成立。C_1 和 C_2 任一成立，而且 C_3 成立时，E_1 成立。C_3 不成立时，E_3 成立；C_3 成立时，E_3 不成立。因为原因 C_1 和 C_2 不可能同时取 1 状态值，所以图中用 E(互斥)符号表示它们之间的互斥关系。

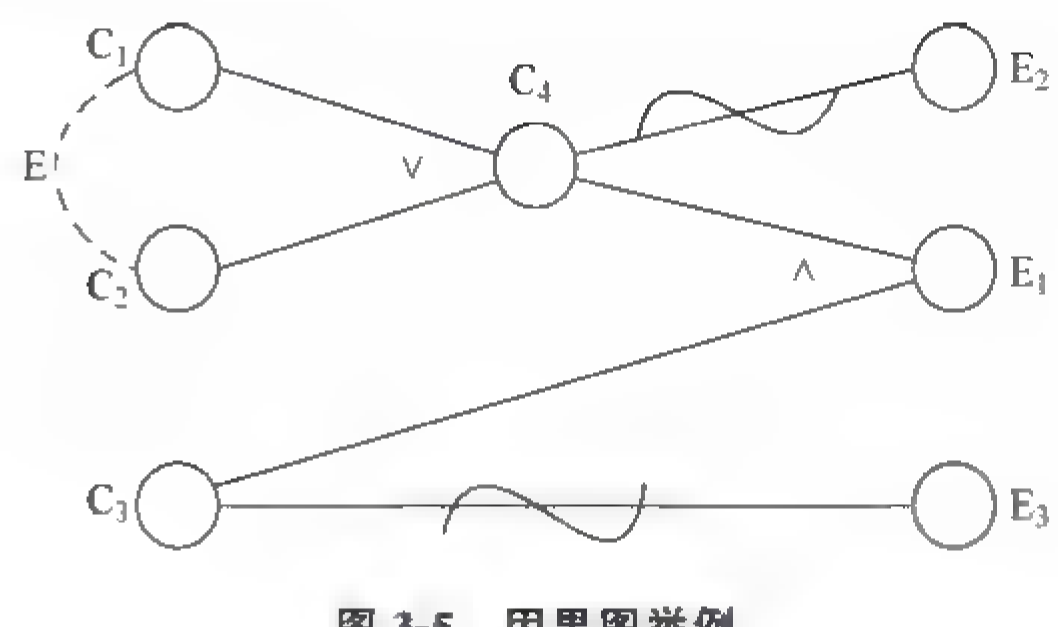


图 3-5 因果图举例

将图 3-5 原因组合及相应的结果组合用表的方式，其间的关系更为明了。在转成表格时，需列举出原因的所有组合及相应的结果组合，但应注意有些原因的组合是不存在的。在图 3-5 中，有 3 个原因，各取状态值 0 和 1，一共有 8 种可能的组合，但由于 C_1 和 C_2 同时取 1 是不可能的，所以只有 6 种组合(见表 3-1)。要确认因果图是否正确表达了规格说明书，应该通过设置原因的所有可能状态并直到将所有结果设置成正确的值。

表 3-1 因果图的列表示例

原因与结果 \ 组合列举		1	2	3	4	5	6
输入(原因)	C_1	0	0	0	0	1	1
	C_2	0	0	1	1	0	0
	C_3	0	1	0	1	0	1
输出(结果)	E_1	0	0	0	1	0	1
	E_2	1	1	0	0	0	0
	E_3	1	0	1	0	1	0

将因果图的列表转成决策表，方法上很直接，只须将因果图的列表中的表项名称转为决策表的表项名称，即在决策表中称原因为条件，结果为行动，原因与结果组合为决策规则，如表 3-2 所示。

表 3-2 决策表示例

条件与行动 \ 决策规则		1	2	3	4	5	6
条件	C_1	0	0	0	0	1	1
	C_2	0	0	1	1	0	0
	C_3	0	1	0	1	0	1
行动	A_1	0	0	0	1	0	1
	A_2	1	1	0	0	0	0
	A_3	1	0	1	0	1	0

所谓测试用例，须提供两种信息：一是输入条件，二是期望的输出结果。决策表中每一个决策规则提供测试用例所需的两种信息，所以决策表中每一个决策规则是一个测试用例。

前面两节介绍的等价类划分法和边界值分析法都是着重考虑输入条件，并没有考虑到输入条件的各种组合，也没有考虑到各个输入条件之间的相互制约关系。当测试必须考虑多个输入条件的各种组合，及其相应的产生多种结果或动作时，可以利用上述因果分析法。

下面总结一下利用因果图、决策表的因果分析法执行过程：

(1) 分析程序规格说明书，识别哪些是原因，哪些是结果。原因往往是输入条件或是输入条件的等价类，而结果常常为输出条件。

(2) 分析程序规格说明书，按其语义，在因果图连接各个原因与其相应的结果。用本节讲述的4种关系符号(—、～、∨、∧)来描述因果图中原因与结果之间的关系。

(3) 标明约束条件。由于语法或环境的限制，有些原因和结果的组合情况是不可能出现的。对于这些特定的情况，在因果图中使用本节讲述的5种约束符号(E、I、O、R、M)来标明原因间、结果间的约束条件。

(4) 把因果图转换成一个因果图列表进而生成决策表(或判定树)来描述哪种输入组合所引起的哪个执行动作的决策规则。

(5) 把决策表的规则转换成测试用例。选择测试数据以便使决策表里的每个规则都被测试。显而易见，如果决策表已被用作设计工具，那么就不必进行因果分析了，可以直接利用设计时所得到的决策表生成测试用例。

3.5 正交数组测试

有很多应用，其输入域是有限的。也就是说，输入参数的数量不多，而且每个参数能取的值也是明显确定的。当这些数量非常小时(例如，3个输入参数，每个参数可取值各为3个离散值)，则可以考虑对输入参数值进行排列组合，即用枚举法详尽地测试所有输入域。然而，随着输入参数的增加和每个输入参数的取值数量的增加，枚举法测试将不再适用。

正交数组测试(Orthogonal Array Testing)可以应用于输入域相对较小而详尽测试的次数又过于巨大的问题。正交数组测试方法对于发现和区域错误(Region Faults)相关的错误特别有用。区域错误是和软件模块中的逻辑相关的一类错误。

为了描述正交数组测试和较传统的“一次一个输入项(one input item at a time)”方法的区别，考虑一个有3个输入项的系统：X、Y和Z。每一个输入项有3个离散值与之相关，那么就有$3^3=27$种可能的测试用例。Phadke[3]建议可以从几何学角度来看这些可能的测试用例，如图3-6所示。图中每次只能有一个输入项沿着其输入轴变化，这只能达到对输入域相对有限的覆盖(如图3-6(a)所示)。

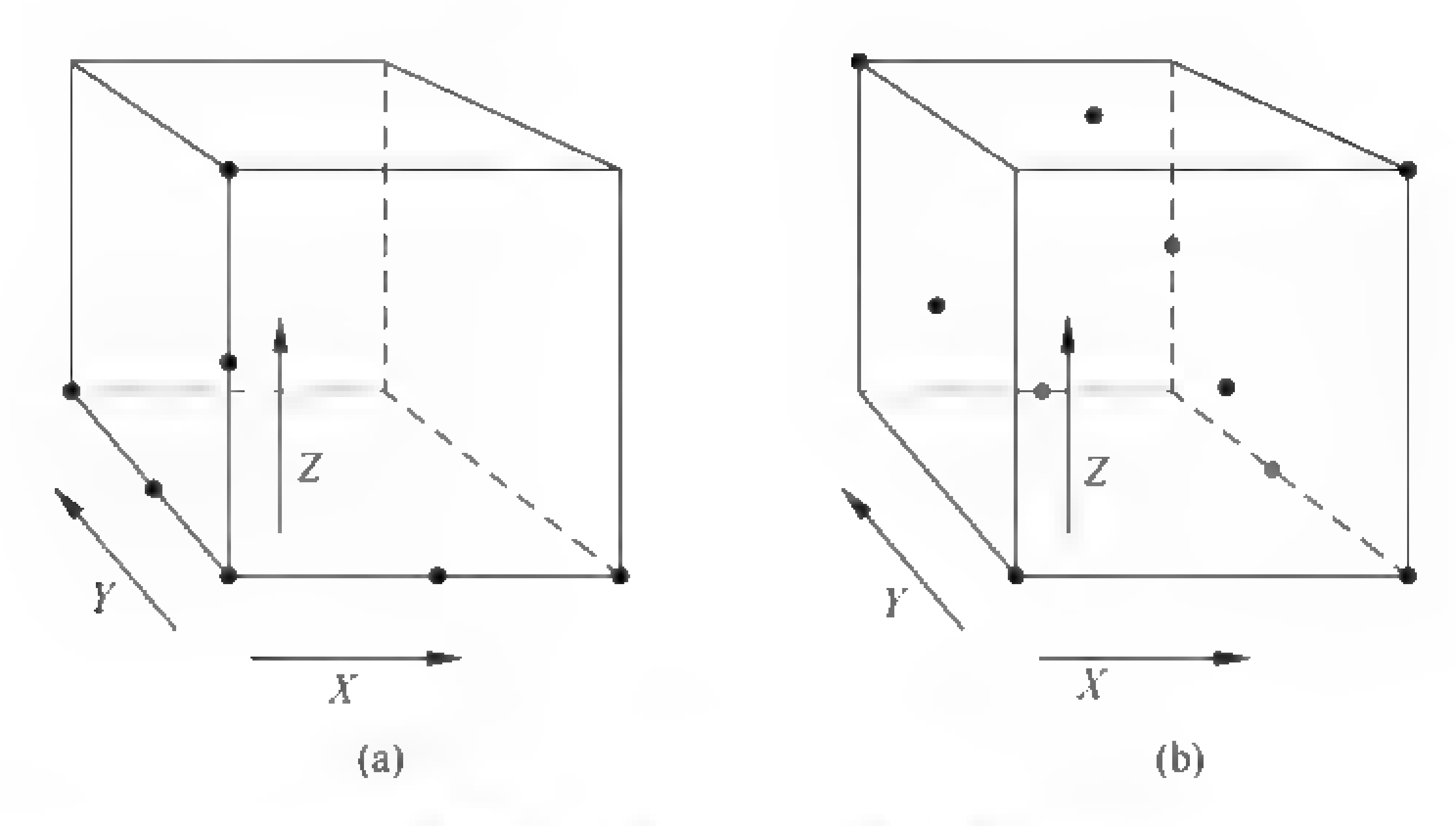


图 3-6 测试用例几何视图

若使用正交数组测试法，则会建立一个测试用例的 L9 正交数组。L9 正交数组具有"平衡特性[3]"，也就是测试用例(图 3-6(b)中黑点所表示的)是"均匀分布在测试域中的"，如图 3-6(b)所示。这对输入域的测试覆盖更为完全。为了说明如何使用 L9 测试数组，举个传真机发送功能(send 函数)的例子。假设有 4 个参数：P_1、P_2、P_3 和 P_4，被传到 send 函数，其中每一个参数都可以取 3 个离散值。例如，P_1 取值分别为：

$P_1=1$，立即发送
$P_1=2$，一小时后发送
$P_1=3$，午夜后发送

P_2、P_3 和 P_4 也可取值为 1、2 和 3。

如果采用"同一时间只改变一个输入项的值"的测试策略，那么测试序列(P_1，P_2，P_3，P_4)将设为：(1,1,1,1)、(2,1,1,1)、(3,1,1,1)、(1,2,1,1)、(1,3,1,1)、(1,1,2,1)、(1,1,3,1)、(1,1,1,2)和(1,1,1,3)。以下是 Phadke[3]对这些测试用例的评估：

这些测试用例只有在测试参数互不影响的情况下有用。当只有一个参数的值造成软件故障时，它们可以查出这样的逻辑错误。这种错误称为单模错误(Single Mode Faults)。这种方法不能查出当两个或多个参数同时取某些值时导致软件故障的逻辑错误，也就是说，它不能检测参数之间互相影响的情况。因此它的查错能力是有限的。

给定相对少量的输入参数和离散值，穷举测试是可能的。上述例子所需的测试的数量是 $3^4=81$，这个数虽然还是比较大的，但尚可管理。所有和数据项排列相关的错误将均可被发现，但是所需的工作量相对较高。

正交数组测试法提供了一种测试覆盖率高，而测试用例数量又远比穷举测试少的测试方法。对于传真机的例子，其 L9 正交数组如表 3-3 所示。

表 3-3　一个 L9 的正交数组

测试用例	测试参数			
	P_1	P_2	P_3	P_4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Phadke[3]对使用 L9 正交数组后的测试结果进行评估：

- 检查并且隔离所有的单模错误。单模错误是指任一参数在某个值的情况下都会出现的错误。例如，如果所有包含 $P_1=1$ 的测试用例都会产生一样的错误情况，那么这就是一个单模错误。本例中测试用例 1、2 和 3(如表 3-3 所示)将会显示错误。通过分析有关哪些测试用例显示错误的信息，可以确认是哪个参数值引起的错误。本例中，因为测试用例 1、2 和 3 引起错误，所以可以把与"立即发送($P_1=1$)"相关的逻辑处理确定为错误源。这种对错误的隔离对修正错误是很重要的。
- 检查所有的双模错误。如果两个参数的某些取值同时出现而引起一种固定的错误，则称为双模错误。一个双模错误暗示了两个参数之间的互不相容或者是互相有害的作用。
- 多模错误。正交数组可能保证检测到单模错误和双模错误。然而，许多多模错误也可由这些测试查找出来。

有关正交数组测试的详尽讨论，可参考文献 Phadke 1989。

3.6　测试插桩

测试插桩(Instrumentation)是指在程序的特定部位附加一些操作或功能，可用来检验程序运行结果以及执行特性，以便支持测试系统。目前有两大类测试插桩技术，分别支持黑盒测试和白盒测试。支持白盒测试的测试插桩技术可提供以下功能：

(1) 生成给定状态或者内部数据表示法以强迫程序进入一个特定的状态，以便检验状态的可达性。

(2) 显示或读取内部数据或者私有数据；这类数据一般不能从程序外部获得。

(3) 监测具体的数据不变特性；程序中的某些数据在执行过程中保持不变，在程序不

同位置插入语句可以观测此数据不变特性。

(4) 监测前提条件：在执行某操作或过程之前，插入语句可以观测操作或过程前提条件。

(5) 监测后置条件：在执行某操作或过程之后，插入语句可以观测操作或过程后置条件。

(6) 人为触发时间事件：当某事件发生所需时间太短或太长，在程序不同位置插入适当功能可以控制时间事件的发生。

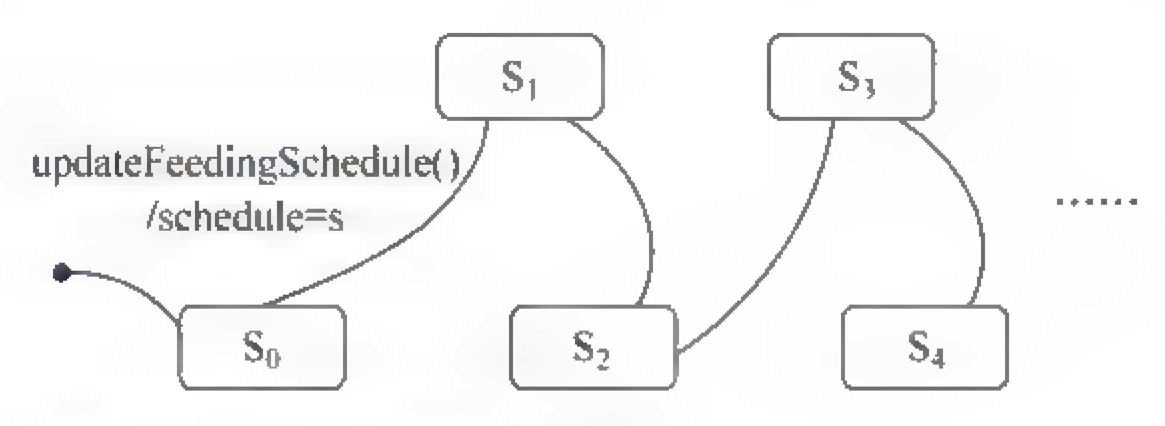
(7) 监测事件时间来测试时间约束：在程序中事件发生前后位置插入语句可以记录事件发生时间以检验时间约束。

本节介绍两种支持黑盒测试的插桩技术：测试预言与随机数生成器。

- 测试预言(Oracle)：一个程序，是用来确定对于给定的系统输入数据或输入序列，被测系统是否有一个特定输出。
- 随机数据生成器：用来产生测试数据的一个程序。

3.6.1 测试预言

测试预言是实现黑盒测试规格说明中的一个模块里的谓词(Predicates)或条件。下面以自动化喂鱼系统中喂鱼视图模块为例，介绍测试预言技术的应用。喂鱼视图模块规定两次喂鱼时间间隔，允许修改喂鱼时间表，但必须满足喂鱼时间规定的间隔。图3-7给出了喂鱼视图状态图的描述。



State(schedule:FeedingSchedule);
Initially at S_0,schedule=null;
Invariant：at all states except S_0,f.start-f.end<=30minutes;
Event message：updateFeedingSchedule();
Action on Transition：schedule=s;

图3-7 状态图模型——喂鱼视图

图3-7中的状态表示系统使用不同的时间表而处于不同的状态，其中处于初始状态 S_0 时，时间表为空，图3-7中省略号表示其他时间表状态；除了初始状态 S_0，处于任一状态时系统都有一个不变量，即时间表里两次喂鱼时间间隔不超过30分钟；当系统接收到修改时间表请求时，系统修改时间表转入下一个状态。图3-8给出了喂鱼视图模块的一个设计类

图及其程序实现。

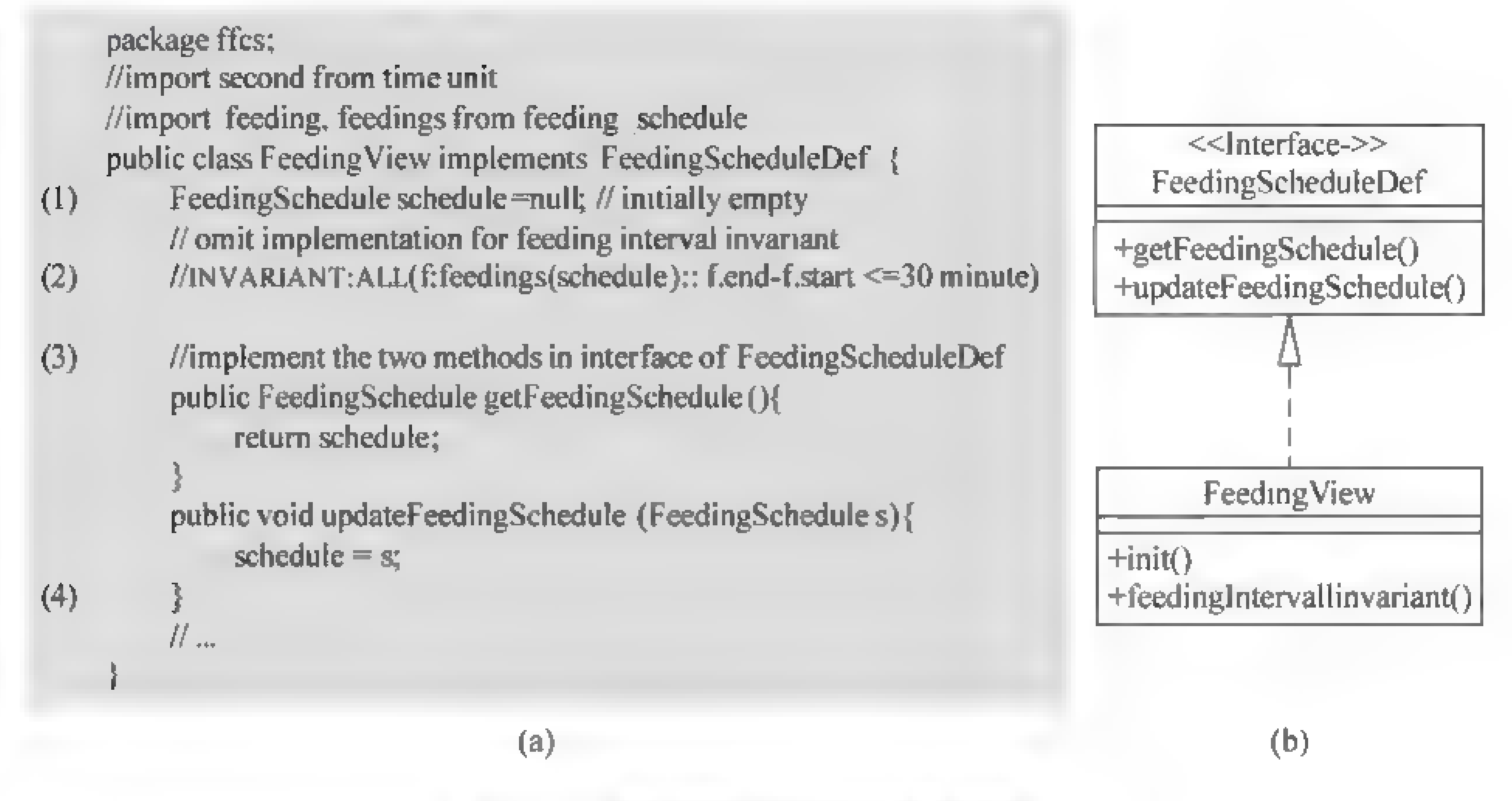


图 3-8 喂鱼视图的类图及实现代码

如图 3-8(b)所示的类图中 FeedingScheduleDef 是接口，定义了两个操作：getFeedingSchedule()和 updateFeedingSchedule()；FeedingView 是类，实现接口的两个操作并定义喂鱼视图的初始条件及两次喂鱼时间间隔不变量。图 3-8(a)是用 Java 语言写的程序。

(1) 程序中标号(1)和(2)间定义了时间表的初始条件及两次喂鱼时间间隔不超过 30 分钟的不变量。

(2) 程序中标号(3)和(4)间实现了接口的两个操作。

图 3-9 是喂鱼视图的测试预言程序 TestFeedingView，其中有 3 个方法：checkInvariant()，testUpdateFeedingSchedule()和 main()。

(1) 为测试程序 FeedingView，需先创建一个实例 fv(见程序中行号(1))。

(2) 程序中行号(2)～(5)间定义了 checkInvariant()。其中调用 FeedingView 实例 fv 的 getFeedingSchedule()得到喂鱼时间表的一个实例 fs，然后比较喂鱼时间表的结束时间与开始时间的间隔是否大于 30 分(程序中将分转为微秒)。如果大于，则返回 false，否则返回 true。checkInvariant()是私有方法，被 testUpdateFeedingSchedule()调用。

(3) 程序中行号(6)～(14)间定义了 testUpdateFeedingSchedule()。其中先创建一个喂鱼时间表实例 fs，利用 Calendar 类得到一个系统当前时间实例 now，并把 now 设为时间表开始时间；把当前时间 now 推进 31 分，在将其设为时间表结束时间。调用 fv 的 updateFeedingSchedule()并把 fs 作为其参数传入，来更改 fv 中喂鱼时间表，返回调用 checkInvariant()的结果。

(4) 程序中行号(15)～(21)间用 main()方法来测试 testUpdateFeedingSchedule()。先创建 TestFeedingView 的一个实例 tfv。如果 tfv.testUpdateFeedingSchedule()返回 true，则显示

"testUpdateFeedingSchedule( ) failed.",否则显示"testUpdateFeedingSchedule( ) succeeded."。按照程序中行号(6)~(14)定义,应显示前者,因为 testUpdateFeedingSchedule()把时间间隔设为 31 分,而不变量是 30 分。

注意:如果测试人员需测试修改 FeedingSchedule 的多种情形,可以写多个不同的 testUpdateFeedingSchedule( )方法,并将其组成套件,在测试预言程序里一同执行测试。JUnit 工具是采用这种方式(见第 6 章)。上述测试预言程序里调用被测类程序中的两个方法,测试预言不属于纯黑盒测试,而是一种灰盒测试(Bray box Testing)。灰盒测试技术利用被测程序的应用程序接口(API)或被测系统的执行路径来进行测试。

```
      public class TestFeedingView {
(1)       FeedingView fv=new FeedingView();
          //检查不变量
(2)       private boolean checkInvariant() {
(3)           FeedingSchedule fs=fv.getFeedingSchedule();
(4)           return (fs.getEnd()-fs.getStart() > 30 * 60 * 1000)? false: true;
(5)       }
          //测试方法 updateFeedingSchedule()
(6)       public boolean testUpdateFeedingSchedule(){
(7)           FeedingSchedule fs=new FeedingSchedule();
(8)           Calendar now=Calendar.getInstance();
(9)           fs.setStart(now);
(10)          now.add(now.MINUTE,31);
(11)          fs.setEnd(now);
(12)          fv.updateFeedingSchedule(fs);
(13)          return checkInvariant();
(14)      }
          //执行测试
(15)      public static void main (String[] args) {
(16)          TestFeedingView tfv=new TestFeedingView();
(17)          if (! tfv.testUpdateFeedingSchedule())
(18)          System.out.println ("testUpdateFeedingSchedule() failed.");
(19)          else
(20)          System.out.println ("testUpdateFeedingSchedule() succeeded.");
(21)      }
      }
```

图 3-9 喂鱼视图的测试预言程序

3.6.2 随机数据生成器

随机测试技术(Random Testing Technique)是一种黑盒测试方法,通过在所有可能的输入值中选取一个任意的子集来对软件进行测试[①]。随机测试有助于避免这样的问题:只

① 参考 URL:http://computing-dictionary.thefreedictionary.com/random+testing 的定义。

测试你所知道的将奏效的场景。1990年Dick Hamlet比较利用等价划分的子域边界测试结果与忽视等价划分的随机测试结果，指出等价划分的作用微乎其微。但完全的随机测试方法难以进行有针对性有目标的测试，从而导致低效测试活动。一般认为，在生成测试用例时，结合使用等价划分、边界值和随机测试3种技术是一种有效方法。即在先将软件输入域进行等价划分，在各个子域边界上及附近选取中边界值，然后再在各个子域里用随机测试方法选择软件输入样本点。

例如一个软件有效输入域是1～100之间的整数，包括1和100。用等价划分法，测试输入域是：

无效域1：输入数据<1
有效域：1≤输入数据≤100
无效域2：输入数据>100

用边界值分析法，可生成3个子域的边界值，例如，分别为{x|x<=0}、{1,2-99,100}和{y|y>=101}。然后在3个子域里用随机测试方法选择软件输入样本点。

这里介绍一下如何用随机数据生成器有效地帮助产生测试所需的输入数据。随机数据生成器是运用了Java API中的public static double random()。调用后返回一个带正号的double值，大于或等于0.0，并且小于1.0，返回值伪随机地均匀地分布在这个范围内。即，0.0<=Math.random()<1.0。若要产生一个1～100之间的随机数，则执行(int)(Math.random()*100)+1。例如：设有一个字符数组char[n]，若是要产生一个长度小于或等于n的字符串，可以用Math.random()来产生一个随机数k。代码如下：

```
⋮
String s="";
int k=(int)(Math.random()*n)+1;
for(int i=0; i<k; i++)s+=char[i];
return s;
⋮
```

随机数据生成器能为执行测试提供方便功能，被认为是一种测试插桩法。本节运用Java API中Math类的random()方法介绍了随机数据生成器的创建。

3.7　总结

黑盒测试是一种不知道被测事物内部逻辑的测试。例如，当黑盒测试应用于软件工程，测试者只知道合法输入和期望输出，而不知道程序是如何实现这些输出。因为黑盒测试可以看作是根据规约的测试，其他关于程序的知识是不需要的。基于这个原因，测试者和编程者彼此独立，避免了编程者有偏好地进行其工作。

黑盒测试注重于测试软件的功能性需求，也即黑盒测试需要软件工程师生成输入条件

集来检测程序所有功能需求。黑盒测试并不是白盒测试的替代品，而是配合白盒测试发现其他类型的错误。黑盒测试试图发现以下类型的错误：

(1) 功能不对或遗漏。

(2) 界面错误。

(3) 数据结构或外部数据库访问错误。

(4) 性能错误。

(5) 初始化和终止错误。

白盒测试在测试的早期执行，而黑盒测试主要用于测试的后期。黑盒测试故意不考虑控制结构，而是注意信息域。

3.8 参考文献

[1] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons. 1 edition, 1979

[2] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Wiley, 1995

[3] M. S. Phadke. Planning Eiifcient Software Tests. *Crosstalk*. vol. 10, no. 10, October 1997, pp. 11～15

[4] M. S. Phadke. *Quality Engineering Using Robust Design*. Prentice-Hall, 1989

3.9 思考与练习

1. 黑盒测试与白盒测试的主要区别是什么？对什么样的错误，你可能用黑盒测试方法去发现？对什么样的错误，你可能用白盒测试方法去发现？

2. 如何使用等价划分技术生成测试用例？试举例说明。

3. 边界值分析方法如何帮助生成测试用例？如何结合使用等价划分技术和边界值分析方法生成测试用例？

4. 给定功能需求：

(1) 如果用户没有被授权提出药品申请，则系统将拒绝他申请新的化学药品。

(2) 如果用户被授权提出药品申请，但他申请的药品或者没有库存，或者无法从供货商获取，则系统也将拒绝他申请新的化学药品。

(3) 如果用户被授权提出药品申请，且被申请药品不存在于有害药品清单中，则系统接受用户申请。

(4) 如果用户被授权提出药品申请，且被申请药品存在于有害药品清单中，但此用户已接受过有害药品处理的相关培训，则系统接受用户申请。

用因果分析方法生成测试用例。

5. 说明什么是"单模"错误,什么是"双模"错误?说明为什么正交数组方法适用于测试"区域错误"?

6. 假设 a、b 是一个类方法的两个参数,$1 \leqslant a \leqslant 100, b \in S=\{A, B, \cdots, Z\}$。写一个 Java 方法来自动产生测试输入。

3.10 进一步阅读

R. Pressman. Software Engineering: A Practitioner's Approach. Boston: McGraw Hill,2005

C. Kaner,J. Bach,B. Pettichord. Lessons Learned in Software Testing. John Wiley & Sons,2002

L. Copeland. A Practitioner's Guide to Software Test Design. Boston: Artech House Publishers,2004

R. D. Craig,S. P. Jaskiel. *Systematic Software Testing*. Norwood. MA: Artech House Publishers,2002

Beizer,Boris. Software Testing Techniques. 2nd Edition. New York. Van Nostrand Reinhold,1990

网站: Center for Software Testing Education & Research - Black Box Software Testing: http://www.testingeducation.org/BBST/index.html,和 BLACK BOX Testing Details: http://www.testingbrain.com/还有很多有关黑盒测试的资料。

第 4 章

软件测试覆盖分析

第 2 章与第 3 章分别介绍了白盒测试与黑盒测试技术。这时可能会提出一个问题，就是“测试执行到何时才是足够的？”我们需要一种方式来知道测试已经执行的程度。测试覆盖是一种可以凭经验确定软件质量的方法。每种测试覆盖意味着一种针对特定种类的程序缺陷的测试技术。

测试覆盖分析可以在测试计划阶段与测试执行阶段进行。在测试计划阶段，须确定用何种测试覆盖分析及相应的覆盖率。覆盖率通常表示为百分比，但是百分比的意义取决于使用了什么测试覆盖分析方法。在测试执行阶段，将根据既定的覆盖率来检查是否进行了足够的测试。基于测试覆盖分析的测试过程可以用图 4-1 表示。生成一组测试用例，用此组用例执行测试，收集测试结果，进行测试覆盖分析，如果测试结果达到既定的覆盖率，停止测试，否则生成附加测试用例，再重复上述过程。

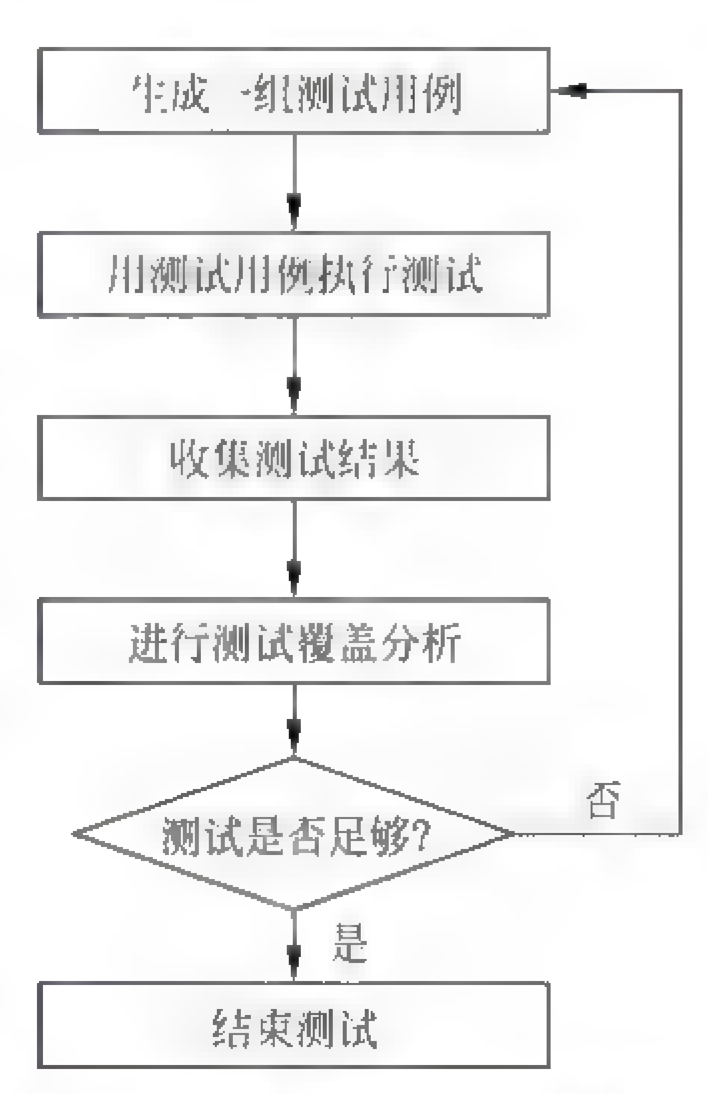


图 4-1　基于测试覆盖分析的测试过程

本章将主要介绍面向白盒测试技术的代码覆盖分析，并简要介绍几种面向黑盒测试技术的覆盖分析方法。

快速阅览：

什么是软件测试覆盖分析？ 软件测试覆盖是一种可以凭经验确定软件质量的方法，每种测试覆盖意味着一种针对特定种类的程序缺陷的测试技术。

由谁来负责软件测试覆盖分析？ 基于白盒测试技术的覆盖分析一般由开发人员进行，基于黑盒测试技术的覆盖分析由有经验的软件测试人员完成。

为什么软件测试覆盖分析如此重要？ 软件测试覆盖分析可以帮助产生合适的测试用

例，并告诉我们测试已经执行的程度。

软件测试覆盖分析步骤各是什么？ 测试覆盖分析可以在测试计划阶段与测试执行阶段进行。在测试计划阶段确定用何种测试覆盖分析及相应的覆盖率，在测试执行阶段根据既定的覆盖率来检查是否进行了足够的测试。基于测试覆盖分析的测试过程是：生成一组测试用例，用这组用例执行测试，收集测试结果，进行测试覆盖分析，如果测试结果达到既定的覆盖率，停止测试，否则生成附加测试用例，再重复上述过程。

有哪些工件形成？ 在测试计划阶段，会生成测试计划和测试用例；在测试执行阶段，会生成测试结果报告和覆盖率报告。

如何确保我们准确地完成了任务？ 在测试执行阶段，根据既定的覆盖率来检查是否完成了足够的测试。

4.1 代码覆盖分析

代码覆盖是测试软件的一种量度标准。它描述程序源代码被测试了的程度。代码覆盖是一种直接观测代码而进行的测试，因而归于白盒测试。代码覆盖技术是最早的系统性软件测试技术中的成员。最早的参考文献是 Miller 和 Maloney 于 1963 年发表在 *Communications of the ACM* 杂志上的[1]。基于代码覆盖的测试的输入是一个程序和一个覆盖标准；输出是一组满足该覆盖标准路径的有限集，称作测试组。基于代码覆盖的测试的两个主要步骤是识别满足覆盖标准的一组实体，然后选择一组覆盖该组实体的有限路径。

有很多种不同的代码覆盖标准及量度代码覆盖的方法，在本节里介绍两种代码覆盖类型：控制流覆盖与数据流覆盖。控制流覆盖是选择一组实体以满足覆盖标准，主要包括语句覆盖、判定覆盖、条件覆盖、多条件覆盖、条件判定组合覆盖、修正条件/判定覆盖及路径覆盖，然后选择一组覆盖该组实体的有限路径。数据流覆盖是选择一组满足变量的定义与引用间的某种关联关系实体，然后选择一组覆盖该组实体的有限路径。无论是哪种覆盖类型，它们都遵循如图 4-2 所示的测试过程。

(1) 由被测程序的源代码，构造程序图。如基本路径法的流图，数据流法的定义使用关联图等。

(2) 根据程序图，生成测试用例。如基本路径法中，先算出环形复杂度，再据此找出基本路径集，生成测试用例。

(3) 编译被测源程序，生成可执行代码(假设源程序无语法错误)。

(4) 生成的可执行代码，用测试用例的输入条件驱动，以执行程序测试。

(5) 计算测试结果的实际覆盖率，如果达不到既定覆盖率，则返回第(2)步，否则结束测试。

(6) 对于测试结果，除了进行代码覆盖分析外，还可以进行其他方面的分析，如测试通过率、失败率、可靠性等。

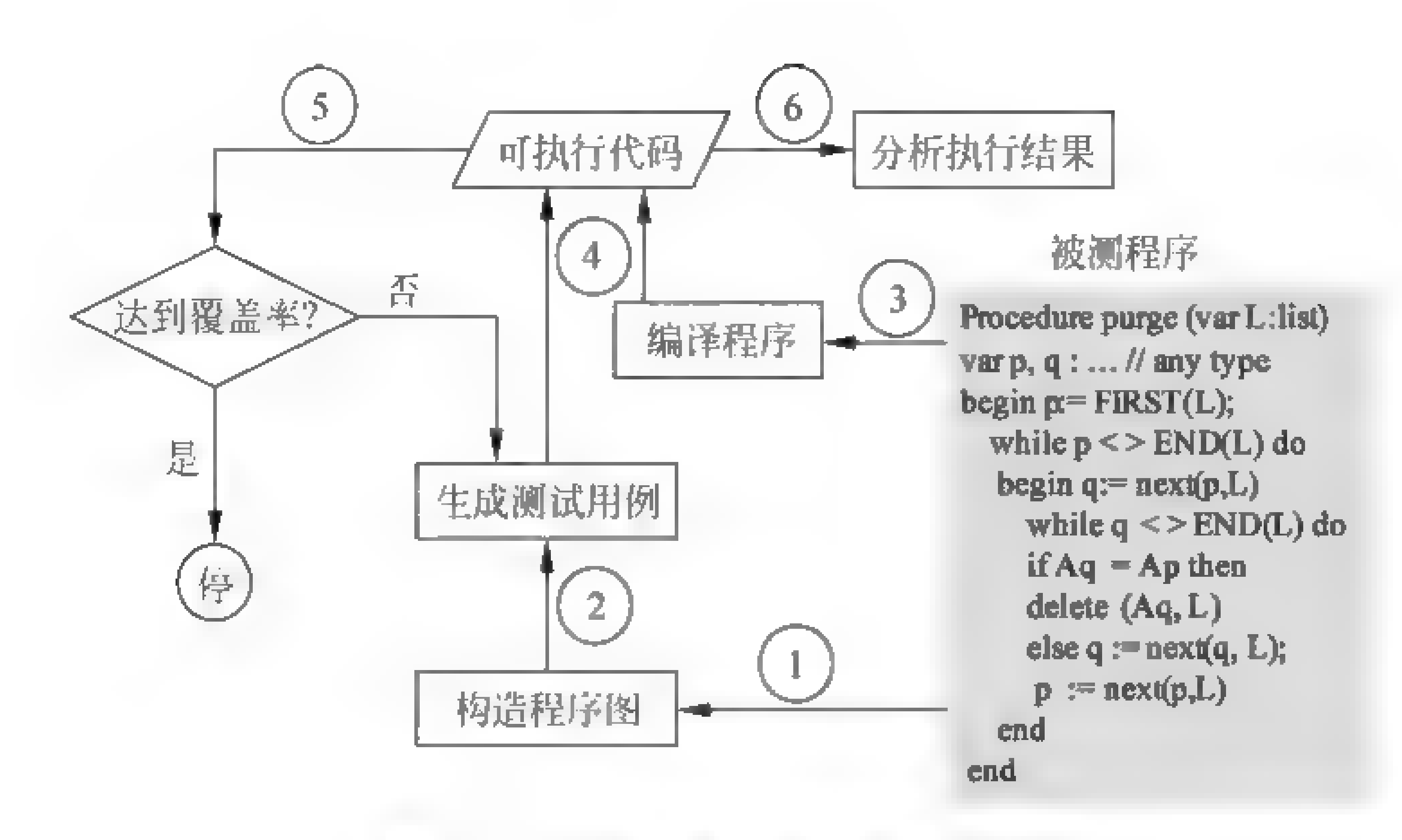


图 4-2 基于代码覆盖分析的测试过程

4.2 控制流覆盖

控制流覆盖是选择一组实体以满足覆盖标准：语句覆盖、判定覆盖、条件覆盖、多条件覆盖、条件判定组合覆盖、修正条件/判定覆盖及路径覆盖，然后选择一组覆盖该组实体的有限路径。

4.2.1 语句覆盖

语句覆盖(Statement Coverage)报告每个可执行语句是否被执行，即每行源代码是否都被执行了并且被测试了。含义是选择足够多的测试数据，使被测程序中每条语句至少执行一次。语句覆盖亦称为线覆盖面(Line Coverage)或段覆盖面(Segment Coverage)[2]。

要达到100%声明覆盖面可能是相当难的。也许有的代码段被设计用来处理错误条件，如果这种错误不出现，这段代码无法执行也就无法测试；或很少发生的事件例如在代码的一个片断接收某一信号，这种情况也给测试此段代码造成困难。也有可能有的代码永远都不会被执行到，例如以下代码段：

```
if ( x > y && y > z && z > x) {
    die "This should never happen!"; //print out the message
}
```

由于if的条件永远为假，所以die显示语句永远得不到执行。这种代码段有时还是有用的，即以某种方式标志这样代码，如果它被执行了，就标识一个错误。这种类型的覆盖是比较弱的，因为即使通过100%语句覆盖的程序也许仍然有严重的问题，而这些问题通过其

他的测试方法可能被发现。看下面的一段代码：

```
int *  p=NULL;
if (condition) {
      p=&variable;
}
 * p=123;
```

对于没有使条件取值为假的一个测试用例，按语句覆盖，这段代码是完全覆盖。实际上，只要条件取值为假，那么这段代码将失败。这是语句覆盖的一个严重的缺点，因为 if 语句在程序中普遍存在。即使如此，对于任何适当大小的软件开发工作，首先使用语句覆盖是可以发现错误的。

语句覆盖的优点是：它可以直接应用于目标代码，并且不需要处理源代码；缺陷是：它对一些控制结构是不敏感的，对程序执行逻辑的覆盖很低，往往发现不了判断中逻辑运算符出现的错误。

语句覆盖计算代码的每一行作为一个可能的语句。这是个好的近似值，但是可能会有各种各样潜在的曲解和滥用，因为理论上每个记号（例如 if）都可以占一行。每一行安置一个记号（这一定不是好的编程样式），代码行数（LOC）可以无谓地加大。同样，程序员可以将多条语句放在同一行中（很明显，这也不是一个好的编程实践），但这样做使得所需测试用例数量大大地减少，因为一行中的任一语句被执行，都认为此行被执行了。

4.2.2 判定覆盖

判定覆盖（Decision Coverage）[3]报告在控制结构中是否测试了布尔表达式取值分别为真和假的情况（例如 if 语句和 while 语句）。整个布尔型的表达式被认为是取值一个 true 和 false，而不考虑内部是否包含了逻辑与（Logical-And）或逻辑或（Logical-Or）操作符。判定覆盖保证只要程序能跳转，它就能跳转到所有可能的目的语句。含义是：设计足够的测试用例，使得每个判定至少都获得一次"真"和"假"，或使得每一个取"真"分支和取"假"分支至少经历一次。判定覆盖亦称为分支覆盖或所有边覆盖。

判定覆盖具有语句覆盖的优点简单性，但是没有语句覆盖所存在的问题。缺点是判定覆盖忽略了在布尔表达式内的分支，这是由短路（Short Circuit）操作符引起的。考虑以下代码段：

```
if (condition1&& (condition2 || function1()))
     statement1;
else
     statement2;
```

这个度量可能认为控制结构被完全地执行，而不考虑对函数 function1 的调用。当

condition1 取值为真和 condition2 取值为真时，测试表达式取值为真；当 condition1 取值为假时，测试表达式取值为假。在这个例子里，短路操作符 || 排除了调用 function1 的影响。

前面已提到判定覆盖的优点是简单的但没有语句覆盖的问题。它是语句覆盖的超集，比语句覆盖的方法要强。判定覆盖的缺点是它忽略了有短路操作符的布尔表达式的分支。当程序中分支的判定由几个条件组合构成时，它未必能发现每个条件的错误。

4.2.3 条件覆盖

条件覆盖(Condition Coverage)报告每个布尔型子表达式的结果是真或假，是否都被执行和测试了。子表达式是用逻辑与运算符和逻辑或运算符分离开的。条件覆盖检查每个判定点(例如真/假的判定)是否被执行和测试了。含义是：构造一组测试用例，使得每一个判定语句中每个子逻辑条件的可能值至少满足一次。考虑以下 C++/Java 代码段：

```
bool f(bool e) { return false; }
bool a[2]={ false,false };
if (f(b && c)) ...
if (a[int (a && b)]) ...
if ((b && c)?false: false) ...
```

上面的 3 个 if 语句不管 b 和 c 的取什么值，控制流都走假分支。这种情况下判定覆盖率只能达到 50%。然而，如果用 b 和 c 所有可能的取值组合来运行这段代码，条件覆盖报告全部覆盖，即条件覆盖率却能达到 100%。条件覆盖与判定覆盖是相似的，但对于控制流有更好的敏感性。完全的条件覆盖并不能保证完全的判定覆盖。

4.2.4 条件判定组合覆盖

条件判定组合覆盖(Condition/Decision Coverage)是条件覆盖(Condition Coverage)和判定覆盖(Decision Coverage)的混合。它有两者的简单性但是没有两者的缺点。条件判定组合覆盖的含义是设计足够的测试用例，使得判定中每个布尔型子表达式条件的所有可能(真/假)至少出现一次，并且每个判定本身的判定结果(真/假)也至少出现一次。

如表 4-1 所示的判定条件 a && (b || c)，选用表中的两组测试用例可以满足条件判定组合覆盖的标准，即第一组中 3 个布尔型子表达式条件 a、b 和 c 都取 T，判定条件 a && (b || c)结果为 T；第二组中 a、b 和 c 都取 F，判定条件 a && (b || c)结果为 F。两组测试用例的判定组合覆盖率为 100%。

表 4-1 条件判定组合覆盖示例

ID	布尔型子表达式			判定条件
	a	b	c	a && (b ‖ c)
1	T	T	T	T
2	F	F	F	F

但是判定组合覆盖也存在一定的缺陷。例如，判定条件 a && (b ‖ c)中的第一个运算符 && 错写成运算符 ‖ 或第二个运算符 ‖ 错写成运算符 && 时，使用表 4-1 中的两组测试用例无法发现这类错误。如表 4-2 所示，用上述两组测试用例测试判定条件 a ‖ (b ‖ c)和判定条件 a && (b && c)，所得到的结果与判定条件 a && (b ‖ c) 一样。虽然上述两组测试用例使得判定组合覆盖率为 100%，但它们无法检查出运算符的错误。

表 4-2 条件判定组合覆盖的缺陷示例

ID	布尔型子表达式			判定条件	
	a	b	c	a ‖ (b ‖ c)	a && (b && c)
1	T	T	T	T	T
2	F	F	F	F	F

4.2.5 多条件覆盖

多条件覆盖(Multiple Condition Coverage)报告每一种可能的布尔型子表达式的组合是否发生了。和条件覆盖一样，子表达式是用逻辑与运算符和逻辑或运算符分离开。它的含义是：生成足够的测试用例，使得每个判定中的条件的各种可能组合都至少出现一次。和条件覆盖一样，多条件覆盖不包括判定覆盖。

对于 Visual Basic 和 Pascal 等不含有短路操作符(Short Circuit Operators)的语言，多条件覆盖实际上是对于逻辑表达式的路径覆盖，和路径覆盖具有相同的优缺点。考虑下列的 Visual Basic 代码段：

```
If a And b Then
...
```

多条件覆盖需要 4 个测试用例，a 和 b 分别取值真(T)和假(F)每一个组合。和路径覆盖相同，每增加一个的逻辑操作符就需要加倍测试用例的数量。一个条件的多条件覆盖所需要的测试用例可用此条件的逻辑操作符的真值表来确定的。如表 4-3 所示的例子，判定语句中有 3 个布尔型子表达式 a、b 和 c，每个子表达式有两种可能取值，T 和 F，因此共有 $2^3=8$种可能组合，表中的 8 个测试用例保证了多条件覆盖率为 100%。

表 4-3 多条件覆盖示例

ID	布尔型子表达式组合			判定条件
	a	b	c	a && (b ‖ c)
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

对于C/C++和Java等具有短路操作符(Short Circuit Operators)的语言，多条件覆盖所要求的一些组合测试执行时无法达到。寻求这样一个最小测试用例集可能是非常冗长乏味的工作，尤其是对于非常复杂的布尔型表达式。多条件覆盖所需要的测试用例的数目对于具有相似的复杂性的条件却有非常大的不同。例如，考虑如下两个C/C++/Java的条件：

(1) a && b && (c ‖ (d && e))
(2) ((a ‖ b) && (c ‖ d)) && e

对于第一个条件，当布尔型子表达式a取F时，由于&&的短路效应，与其后子表达式组合不予进行；当a取T、b取F时，情况相同。而当a取T、b取T并且c取T时，‖短路。如图4-3所示，第一个条件最小测试用例集的势为6。

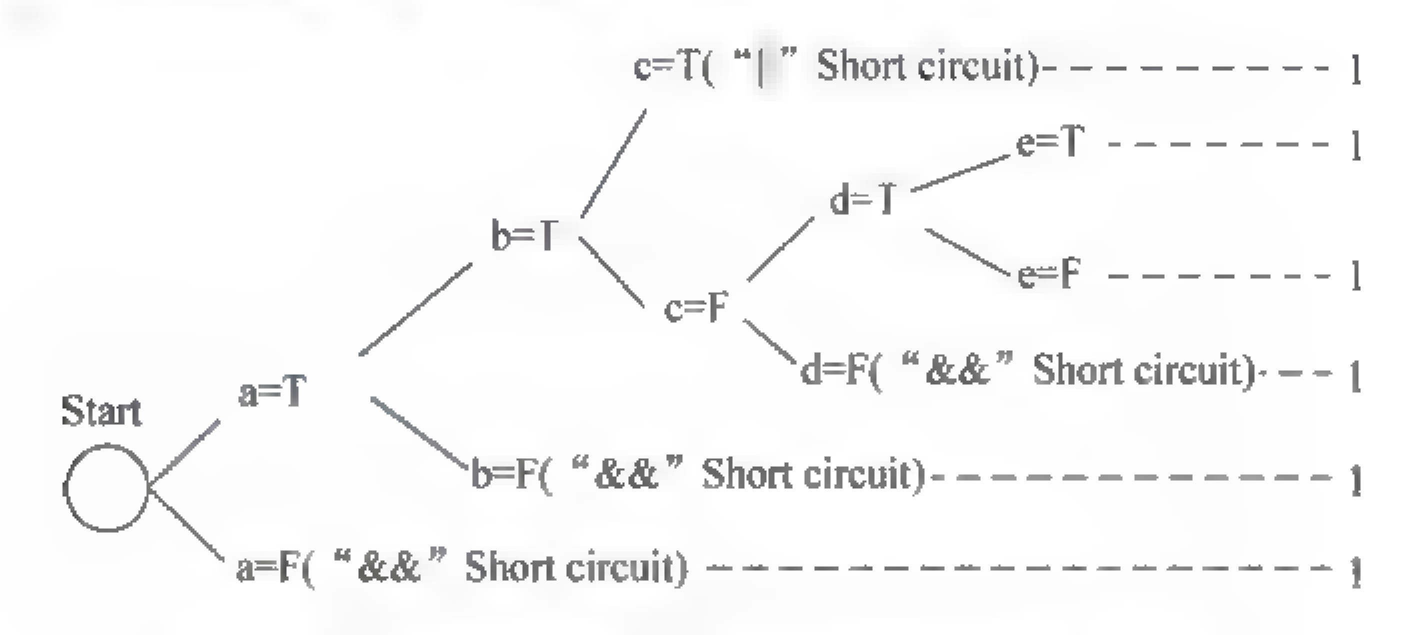


图 4-3 多条件覆盖——具有短路操作例(1)条件

对于第二条件，当布尔型子表达式a取T时，由于‖的短路效应，和b组合不予进行；当c取T时，由于‖的短路效应，和d组合不予进行；当a取F并且b取F时，由于&&的短路效应，与其后子表达式组合不予进行。如图4-4所示，第二个条件最小测试用例集的势为11。

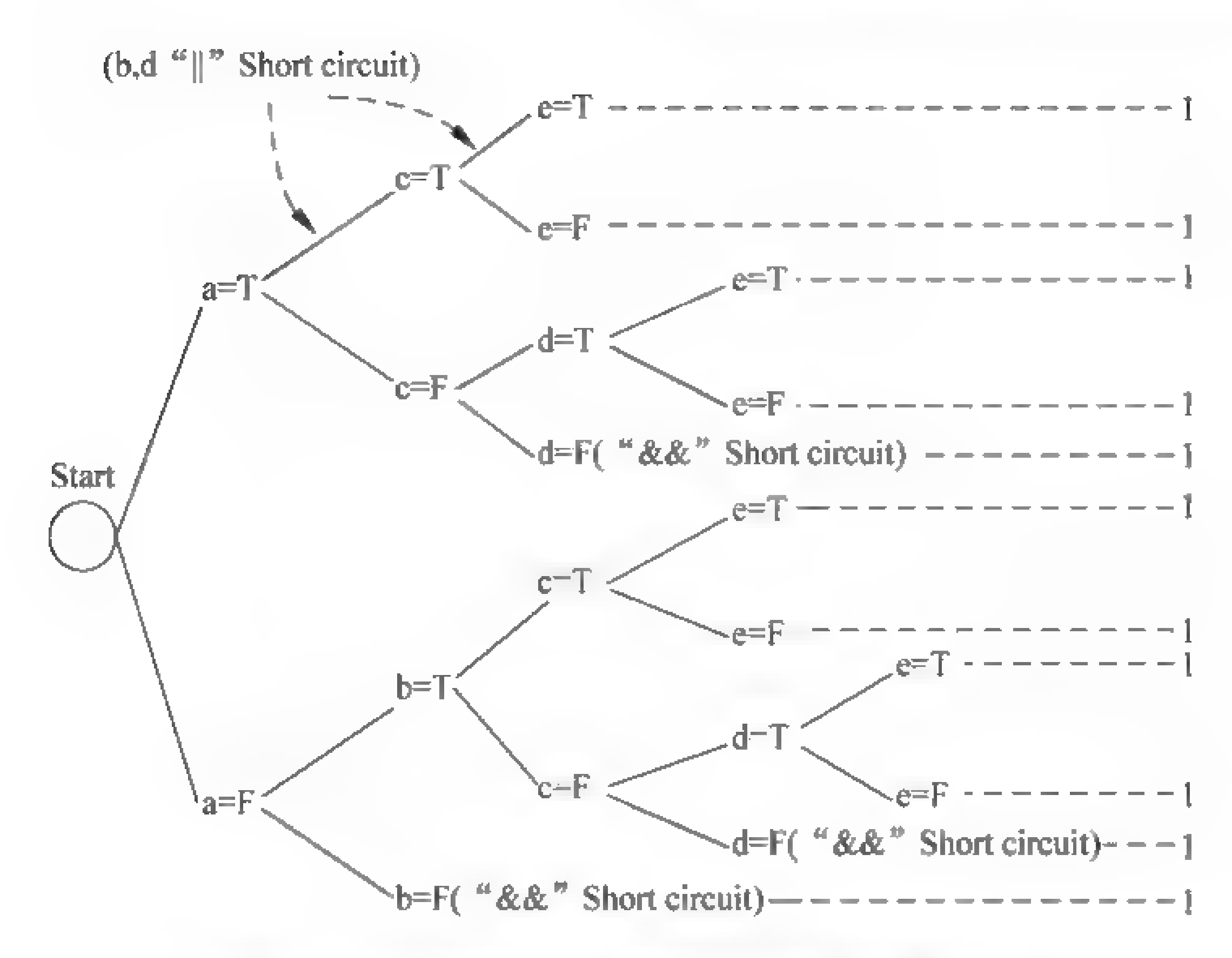


图 4-4 多条件覆盖——具有短路操作例(2)条件

注意，上述两个条件有相同的操作数和操作符，由于短路操作符起作用，要达到完全的多条件覆盖，第一个条件需要 6 个测试用例，而第二个条件需要 11 个测试用例。

4.2.6 修正条件/判定覆盖

修正条件/判定覆盖(Modified Condition/Decision Coverage)也被称为 MC/DC 或 MCDC[4]，最早由波音公司创建。当前在欧美地区，MC/DC 是"空运系统与设备认证软件考虑事项"(RCTA/DO-178B)中必要的测试方法。它要求生成足够测试用例以满足以下 4 种条件：

(1) 程序中的每个入口和出口都要至少被调用一次，即要从每一模块入口至少进入一次并至少要从每一模块出口退出一次。

(2) 程序中每一个判定的所有可能结果至少取一次。

(3) 程序中一个判定的每一个条件的所有可能结果至少取一次。

(4) 程序中一个判定的每一个条件呈现出独立地影响该判定的结果，即只变化此条件而保持其他所有可能的条件不变。

下面用真值表举例说明 MC/DC。表 4-4 所示，对于判定 a && (b || c)，子表达式的可能组合共有 8 种。在测试用例 1 和例 5 中，在 b 和 c 组合不变的情况下，布尔条件 a 的变化独立地引起判定结果的变化，从而布尔条件 a 满足 MC/DC 的要求。对于 a 测试用例 2 和例 6、测试用例 3 和例 7 是同样的情况。在测试用例 2 和例 4 中，在 a 和 c 组合不变的情况下，布尔条件 b 的变化独立地引起判定结果的变化，从而布尔条件 b 满足 MC/DC 的要求。

在测试用例3和例4中，在a和b组合不变的情况下，布尔条件c的变化独立地引起判定结果的变化，从而布尔条件c满足MC/DC的要求。因此要使判定a && (b || c)满足MC/DC的要求，可选定测试用例集{1,2,3,4,5}。显然，满足这类要求地测试用例集可能是不唯一的。

表 4-4 多条件覆盖示例

ID	布尔型子表达式组合			判定条件	各条件独立影响		
	a	b	c	a && (b || c)	a	b	c
1	T	T	T	T	5		
2	T	T	F	T	6	4	
3	T	F	T	T	7		4
4	T	F	F	F		2	3
5	F	T	T	F	1		
6	F	T	F	F	2		
7	F	F	T	F	3		
8	F	F	F	F			

4.2.7 路径覆盖

路径覆盖(Path Coverage)报告是否每个函数的每一条可能的路径都被走过。它检查代码中给定部分每条可能的路径是否都被执行了并且被测试了。一条路径是从函数的入口到出口分支的一个唯一序列。

路径覆盖的一个好处是进行非常彻底的测试。它比判定覆盖方法强。但有两个缺点：一是路径是以分支的数增加而指数级增加，例如：一个函数包含10个if语句，就有$2^{10}=1024$个路径要测试。如果加入一个if语句，路径数就是原来的2倍即$2^{11}=2048$。二是许多路径由于数据相关不可能被执行。考虑以下代码段：

```
if (success)
    statement1;
statement2;
if (success)
    statement3;
```

路径覆盖认为这个程序片段包含4条路径，因为第一个if有两个分支，第二个if也有两个分支，其组合为4。由于两个if的判定表达式完全相同，即都是success，当第一个if取success=true分支时，第二个if也取success=true分支。这样从代码入口到出口只有一条路径。同样，第一个if取success=false分支时，第二个if也取success=false分支，从代码入口到出口也只有一条路径。实际上仅有2条路径而不是4条。

对于一个循环内包含分支的一段程序，从代码中列举所有可能的路径也许是不可能的，

因为路径的数量往往很多。路径覆盖只考虑一个有限数量路径。处理循环的方法有很多种。例如,内部边界路径测试(Boundary-Interior Path Testing)只考虑循环的两个可能性:零次重复和多于零次的重复。对于 do-while 循环有两种可能性:一次反复和多于一次的反复。

研究者提出了很多种路径覆盖技术来避免大数量的路径。例如,长度为 n 的子路径覆盖(n Length Sub Path Coverage)报告是否执行了长度为 n 的每个路径的分支。其他变种包括线性的代码顺序和跳转覆盖(Linear Code Sequence And Jump (LCSAJ) Coverage)和数据流覆盖(Data Flow Coverage)。

4.3 数据流覆盖

用控制流覆盖出现的问题:当语句或分支覆盖被用作测试数据的主要依据时,经过所选的路径并不能保证所有错误都被查出。而且,当路径覆盖标准被用作测试数据的主要依据时,带有循环的程序将有无穷多条路径。一个实用路径的选择必须是依据所用的路径标准,选择一个仅有限数量的路径,发现错误的可能性很大。

数据流覆盖是路径覆盖的一个变异。这类路径覆盖的变异仅考虑从变量定义到其后变量的引用之间的子路径。数据流覆盖中的变量"定义"是指变量赋值而不是类型定义。下面介绍 4 种数据流覆盖选择标准:Rapps 和 Weyuker 的标准、Ntafos 的标准、Ural 的标准及 Laski 和 Korel 的标准。

- Rapps 和 Weyuker 的标准,关注怎样将变量同取值联系在一起以及这些变量是怎么被使用的。
- Ntafos 的标准,使用必需的 k 元组(Required K-tuple),数据流信息去克服单独使用控制流信息选择路径的缺点。
- Ural 的标准研究来自环境的输入对环境的输出的影响。
- Laski 和 Korel 的标准,与选择的次级路径和变量定义/应用的各种各样组合共同到达结点。

这 3 个路径选择的标准是基于数据流分析的。它们之间的关系是:Ntafos 的标准没有要求 k 元组,即包含在 Rapps 和 Weyuker 的标准中定义的所有与定义有关的标准。Ural 的标准以及 Laski 和 Korel 的标准不能满足所选的覆盖了所有边和所有点路径的最小要求。

4.3.1 Rapps 和 Weyuker 的标准

Rapps 和 Weyuker[5]把变量的出现分类为 3 种:定义的(Definitional),用 def 表示;计算使用(Computation Use),用 c-use 或 cuse 表示;和谓语使用(Predicate Use),用 p-use 或 puse 表示。如图 4-5 所示,左面是 getMaxValue 方法的程序,右面是相应的程序图。程序

图里标注：在 v_0 处定义变量 x 和 y；在 v_1 处定义变量 max 并计算使用变量 x 的值；在 v_2 处谓语使用(puse)变量 x 和 y 的值；在 v_3 处重定义变量 max 并计算使用(cuse)变量 y 的值；在 v_4 处计算使用(cuse)变量 max 的值。注意，v_0 语句中变量 x 和 y 被认为是"定义"了，而 v_0 和 v_1 之间的语句"int max"不被看作是定义。因为前者在方法被调用是赋值，而后者只是类型说明并无赋值操作。

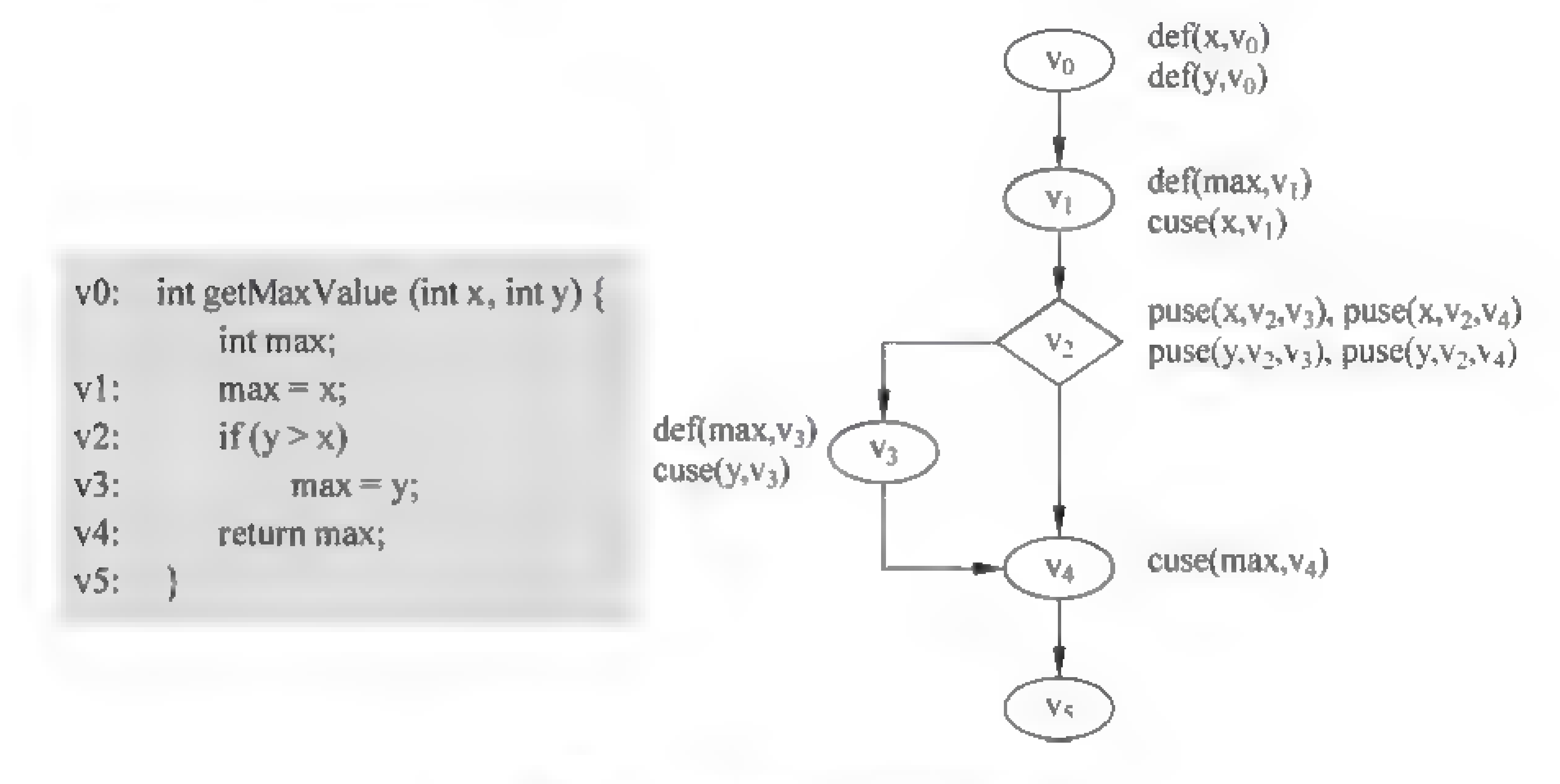


图 4-5 def、cuse 和 puse 示例

下面给出有关 Rapps 和 Weyuker 标准的相关定义。

设 def(x,v)是变量 x 在 v 语句处的一个定义，cuse(x,v')是变量 x 在 v'语句处的一个计算使用。如果 def(x,v)到达 v'，称(def(x,v)，cuse(x,v'))是一个"定义-计算使用对"(definition-c-use pair，dcu-pair)。如果一个"定义清纯"(Definition Clear)的路径包含(def(x,v)，cuse(x,v'))，称此路径覆盖(def(x,v)，cuse(x,v'))的"定义-计算使用对"(dcu-pair)。在如图 4-5 所示的程序图里，(def(max，v_1)，cuse(max，v_4))是一个"dcu-对"。(v_1，v_2，v_4)是一个定义清纯路径；而(v_1，v_2，v_3，v_4)不是，因为在 v_3 语句处变量 max 被重定义。相应的，(v_0，v_1，v_2，v_4，v_5)是覆盖该"dcu-对"的一条路径，而(v_0，v_1，v_2，v_3，v_4，v_5)不是。

设 def(x,v)是变量 x 在 v 语句处的一个定义，puse(x,v',v'')是变量 x 在 v'语句和 v''语句处的一个谓语使用。如果 def(x,v)到达 v'和 v''，称(def(x,v)，puse(x,v',v''))是一个"定义-谓语使用对"(definition-p-use pair，dpu-pair)。如果一个"定义清纯"的路径包含(def(x,v)，puse(x,v',v''))，称此路径覆盖(def(x,v)，puse(x,v',v''))的"定义-谓语使用对"(dpu-pair)。在图 4-5 中的程序图里，(def(x，v_0)，puse(x，v_2，v_3))是一个"dpu-对"，(v_0，v_1，v_2，v_3，v_4，v_5)是覆盖该"dpu-对"的一条路径。(def(x，v_0)，puse(x，v_2，v_4))是另一个"dpu-对"，(v_0，v_1，v_2，v_4，v_5)是覆盖该"dpu-对"的一条路径。

为什么要定义 du 对？图 4-6(a)是"正确"程序，而图 4-6(b)是"不正确"程序。左下面程序的错误是因为在 v_1 处将 max 定义为 x+1。路径(v_0，v_1，v_2，v_3，v_4，v_5)不能测出图 4-6(b)程

序的错误，因为在 v_3 处 max 被重定义，max 作为 getMaxValue 方法的返回值不受上述错误定义的影响。路径(v_0,v_1,v_2,v_3,v_4,v_5)上不存在关于变量 max 的 du 对。路径(v_0,v_1,v_2,v_4,v_5)能测出图 4-6(b)程序的错误，因为在 v1 处 max 的错误定义，直接影响 max 作为 getMaxValue 方法的返回值。路径(v_0,v_1,v_2,v_4,v_5)上(def(max,v_1),cuse(max,v_4))是变量 max 的 du-对。

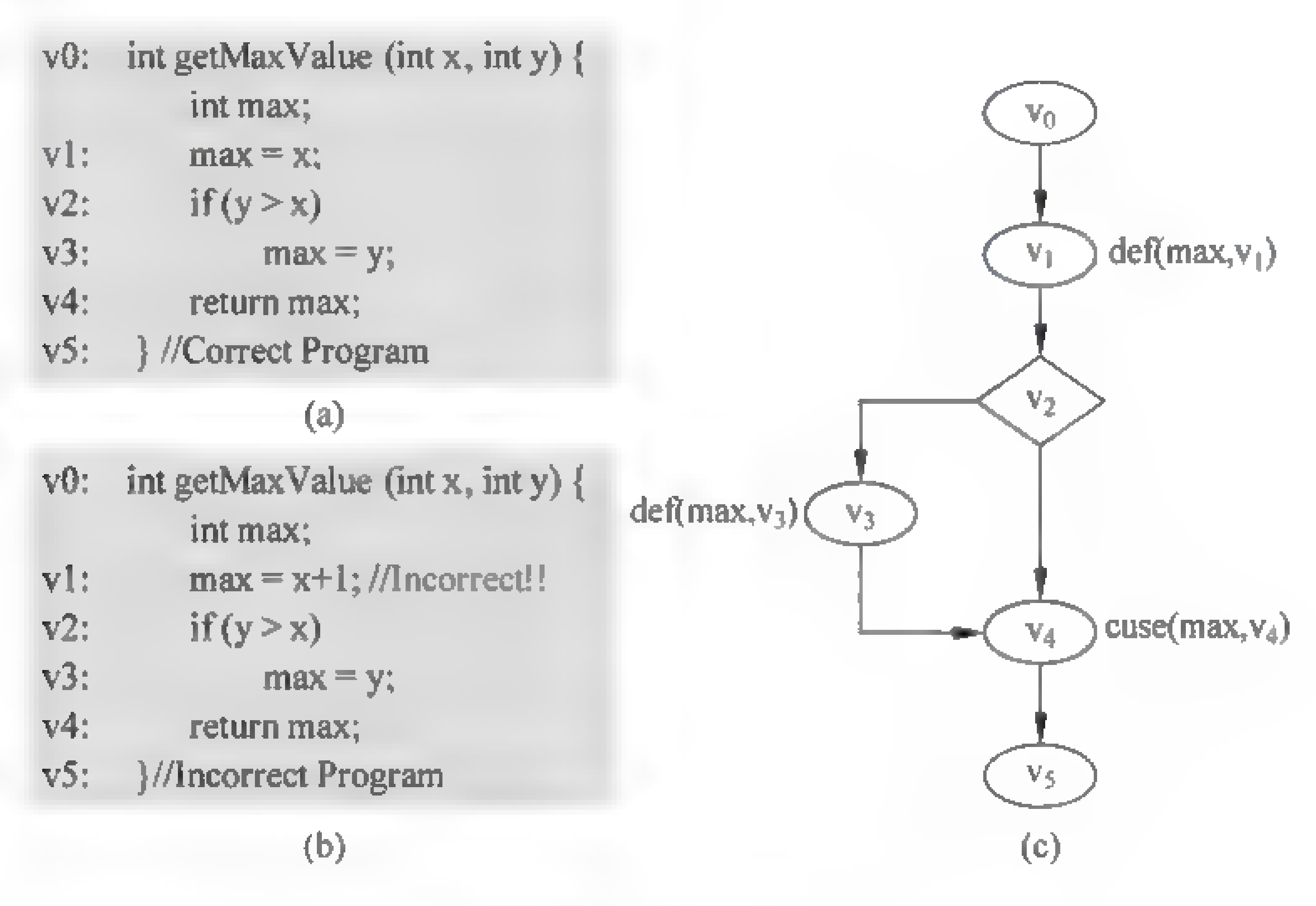


图 4-6　定义 du-对

一个测试组(Test Suite)被称作是满足 all-uses 覆盖标准的：对于每一个变量 x 的每一个定义与每一个使用，如果该定义-使用对是 du-对，则此 du-对被测试组的某一条路径覆盖。一个测试组被称作是满足 all-defs 覆盖标准：对于每一个变量 x 的每一个定义与某个使用，如果该定义-使用对是 du-对，则此 du-对被测试组的某一条路径覆盖。

在如图 4-5 所示的例子中，路径 $P_1=(v_0,v_1,v_2,v_3,v_4,v_5)$ 和路径 $P_2=(v_0,v_1,v_2,v_4,v_5)$ 是满足 all-uses 覆盖标准的一个测试组。如表 4-5，路径 P_1 和 P_2 覆盖了所用变量 x、y 和 max，每一个变量的每一个定义的每一个使用(表中♥符号表示 du 对被 P_1 覆盖；♠符号表示 du 对被 P_2 覆盖)。路径 P_1 是满足 all-defs 覆盖标准的一个测试组，因为路径 P_1 覆盖了所用变量 x、y 和 max，每一个变量的每一个定义的某些使用而不是所有使用。

表 4-5　all-uses 和 all-defs 覆盖标准示例

所有变量每个 du-对 \ 覆盖标准		all-uses		all-defs
		P_1	P_2	P_1
变量 x	(def(x,v_0),cuse(x,v_1))	♥	♠	♥
	(def(x,v_0),puse(x,v_2,v_3))	♥		♥
	(def(x,v_0),puse(x,v_2,v_4))		♠	

续表

所有变量每个du-对 \ 覆盖标准		all uses		all defs
		P_1	P_2	P_1
变量 y	(def(y,v_0),puse(y,v_2,v_3))	♥		♥
	(def(y,v_0),puse(y,v_2,v_4))		♠	
	(def(y,v_0),cuse(y,v_3))	♥		♥
变量 max	(def(max,v_1),cuse(max,v_4))		♠	
	(def(max,v_3),cuse(max,v_4))	♥		♥

下面总结数据流覆盖已提出的路径选择的一组标准，包括 all-paths、all-edges、all-nodes、all-defs、all-uses、all-puses、all-puses/some-cuses 和 all-du-paths（du 代表定义和使用）标准。

- all-paths 标准要求选择一个路径集合包括贯穿流图的每条路径，它对应于路径覆盖。
- all-edges(all-branches 分支覆盖）和 all-nodes(all-statements 语句覆盖）标准要求所选择的路径集合包含流图中的每个边和每个节点。
- all-defs 标准要求选择一个路径集合至少要覆盖从每个定义到该定义的某一使用之间的一个定义清纯子路径。
- all-uses 标准要求选择一个路径集合至少要覆盖从每个定义到该定义的每个使用之间的一个定义清纯子路径。
- all-puses 标准要求选择一个路径集合至少要覆盖从每个定义到该定义的每个谓词使用之间的一个定义清纯子路径。
- all-puses/some-cuses 标准要求选择一个路径集合至少要覆盖从每个定义到该定义的每个谓词使用之间一个定义清纯的子路径。如果定义仅到达计算使用，那么路径集合必须至少包含从每个定义到一个计算使用之间的一个定义清纯子路径。
- all-cuses/some-puses 标准要求选择一个路径集合至少要覆盖从每个定义到该定义的每个计算使用之间一个定义清纯的子路径。如果定义仅到达谓词使用，那么路径集合必须至少包含从每个定义到一个谓词使用之间的一个定义清纯子路径。
- all du paths 标准比 all uses 更进一步。它要求选择一个路径集合要覆盖从每个定义到该定义的每个使用之间的一个定义清纯子路径，并且要求这样的定义清纯子路径是一个简单循环或是无循环的。

路径选择的一组标准间的关系可以用层次图表示。处于层次上方的覆盖标准“强于”处于层次下方的覆盖标准。一个测试标准或策略 X 涵盖某个标准 Y，表示由 Y 产生的测试用例集包含于由 X 产生的测试用例集。图 4 7 给出了覆盖标准的涵盖关系（对于图的解释留给读者来完成）。

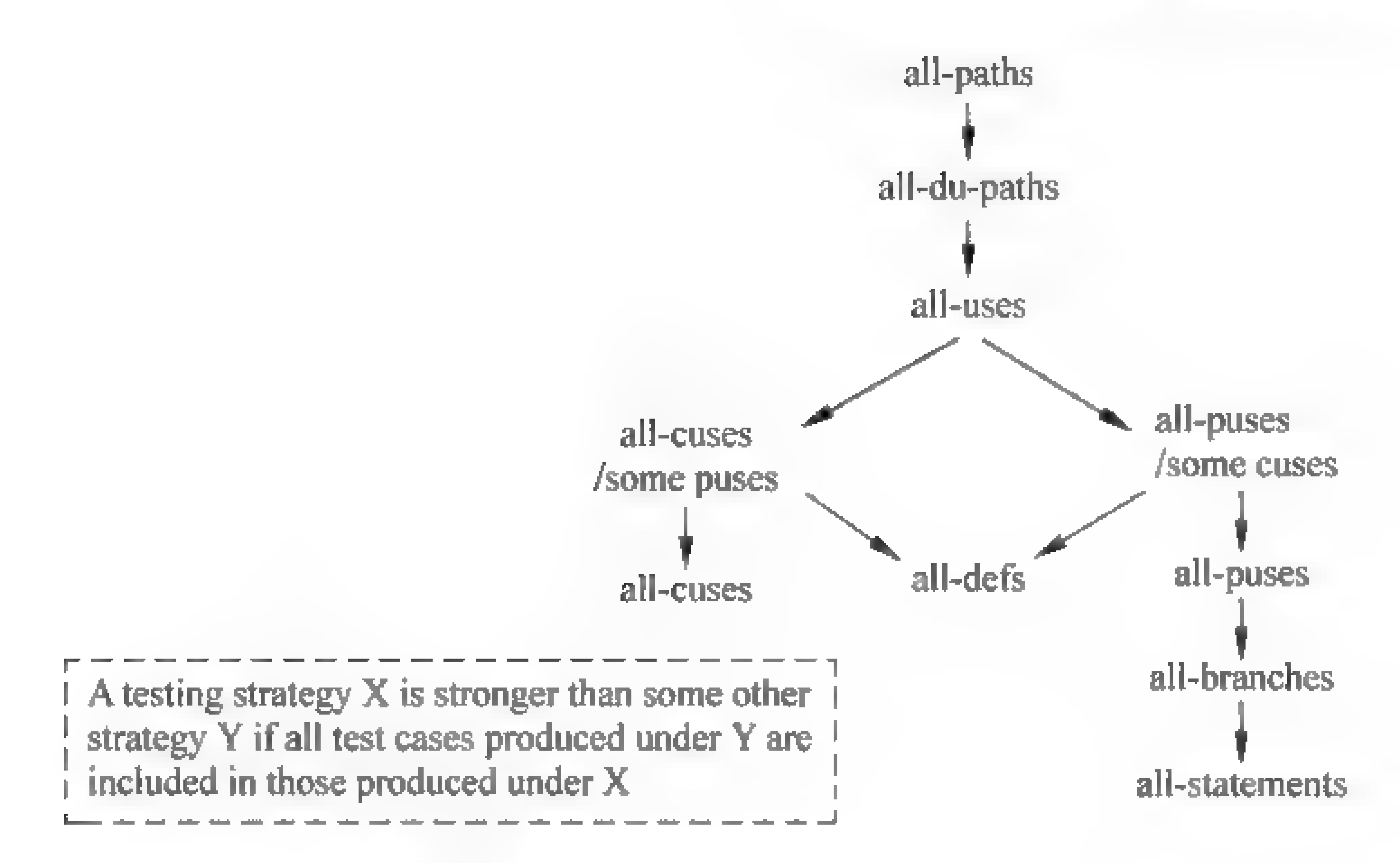


图 4-7　路径选择的一组标准的关系

4.3.2　Ntafos 的标准

在 Ntafos 的标准中定义了数据流链(Data Flow Chain,df-chain)：一个由一组 du-对组成的序列[$(d(x_1,v_1),u(x_1,v_2)),\cdots,(d(x_i,v_i),u(x_i,v_{i'})),(d(x_{i+1},v_{i+1}),u(x_{i+1},v_{i+1'})),\cdots,(d(x_{n+1},v_{n+1}),u(x_{n+1},v_{n+1}))$]，其中 $v_{i'}=v_{i+1}$，即 $v_{i'}$ 和 v_{i+1} 是相同的节点；除了 $u(x_{n+1},v_{n+1})$ 可能是一个谓词使用外，其他都是计算使用。

在如图 4-5 所示的程序图里，$(def(x,v_0),cuse(x,v_1))$ 是一个 du-对，$(def(max,v_1),cuse(max,v_4))$ 是另一个 du-对。这两个 du-对组成一条数据流链(df-链)[$(def(x,v_0),cuse(x,v_1)),(def(max,v_1),cuse(max,v_4))$]。

为什么是一组 du-对组成的一个序列(df 链)而不是一个 du-对？如图 4-8 所示，变量 x 分别在 v_2 和 v_3 节点被定义，各自可能在 v_5 或 v_6 节点被计算使用(cuse)；而这种 cuse 可能是在 v_5 或 v_6 节点用于定义变量 y，进而变量 y 在 v_8 或 v_9 节点被计算使用(cuse)。即在 v_2 节点变量 x 的定义，可能在 v_5 节点通过变量 x 的计算使用(cuse)来定义变量 y，进而影响在 v_9 节点变量 y 的计算使用结果。如果仅关注 du 对而不是 df 链，就不能分析出这类相关影响。

如果一条路径是串接 df 链的每一个 du 对的某“定义清纯”路径，称此路径覆盖该 df 链。如果含有 $k-1$ 个 du 对的每一条 df 链都被一个测试组的某一条路径覆盖，则该测试组(Test Suite)被称作是满足“必须的 k 元组(k Tuples)”覆盖标准。在图 4-5 中的程序图里，两条 df 链[$(def(y,v_0),cuse(y,v_3)),(def(max,v_3),cuse(max,v_4))$]和[$(def(x,v_0),cuse(x,v_1)),(def(max,v_1),cuse(max,v_4))$]分别被路径 $P_1=(v_0,v_1,v_2,v_3,v_4,v_5)$ 和 $P_2=(v_0,v_1,v_2,v_4,v_5)$ 覆盖，由路径 P_1 和 P_2 组成的测试组满足“必须的 3 元组(Required 3 Tuples)”覆盖标准。

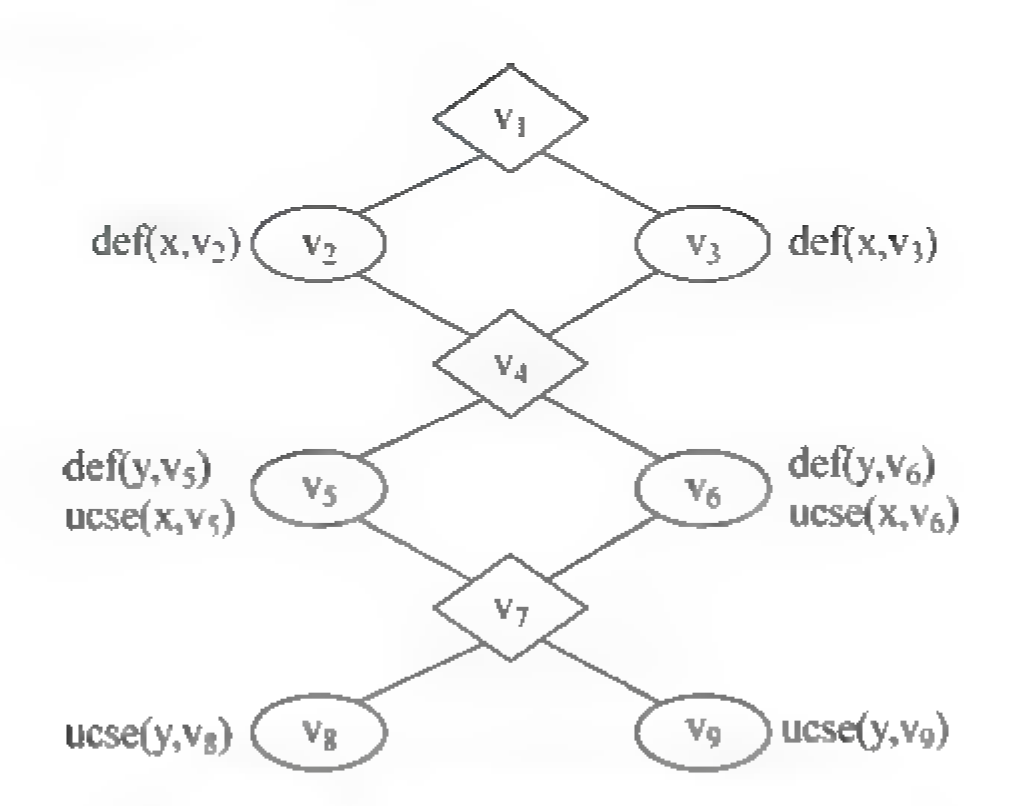


图 4-8 df-链显示的相关影响

注意，Ntafos 的标准没有"必须的 k-元组"来包含在 Rapps 和 Weyuker 的标准中定义的 all-defs 标准。

4.3.3 Ural 的标准

和其他标准不同，基于 Ural 的覆盖标准的测试输入是一个输入参数或全局变量的一个定义；输出是一个输出参数或全局变量的一个使用或是在返回语句里的一个使用。IO-df-链(IO-df-chain)是从输入到输出的一条 df-链，它通过描述来自环境的输入是如何影响对环境的输出，来捕获程序的行为[6]。

Ural 的覆盖标准明确要求从程序传入参数或定义一个全局变量开始到程序输出使用或返回使用。一个测试组(Test Suite)被称作是满足 IO-df-链覆盖标准：如果对于每一个输入和输出，从该输入到该输出之间所有 du-链被测试组的某一条路径覆盖。如图 4-9 所示，输入参数是 x 和 y 的定义，输出是在返回语句里返回 max 的值。路径 $P_1=(v_0,v_1,v_2,v_3,v_4,v_5)$ 和 $P_2=(v_0,v_1,v_2,v_4,v_5)$ 覆盖了从每一个输入(x 和 y)到输出(max)之间的所有 du-链[(def(y,v0),cuse(y,v3)),(def(max,v_3),cuse(max,v_4))]和[(def(x,v_0),cuse(x,v_1)),(def(max,v_1),cuse(max,v_4))]。

4.3.4 Laski 和 Korel 的标准

设 cuse(x_1,v),…,cuse(x_n,v)是一个在节点 v 的使用集合。对于节点 v 的一个有序定义上下文(Ordered Definition Context，ODC)是一个定义序列[def(x_1,v_1),…,def(x_n,v_n)]，使得存在一条路径 v_1 p_1… v_i p_i… v_n p_n v，满足对于每一个 $1\leqslant i\leqslant n$，v_i p_i… v 是关于变量 v_i 的一条"定义清纯"路径。用同样方法，可以定义对于弧(v,v')的一个有序定义上下文。图 4-5 中的程序图里，[def(x,v_0),def(y,v_0)]是弧(v_2,v_3)的一个有序定义上下文(ODC)[7]。

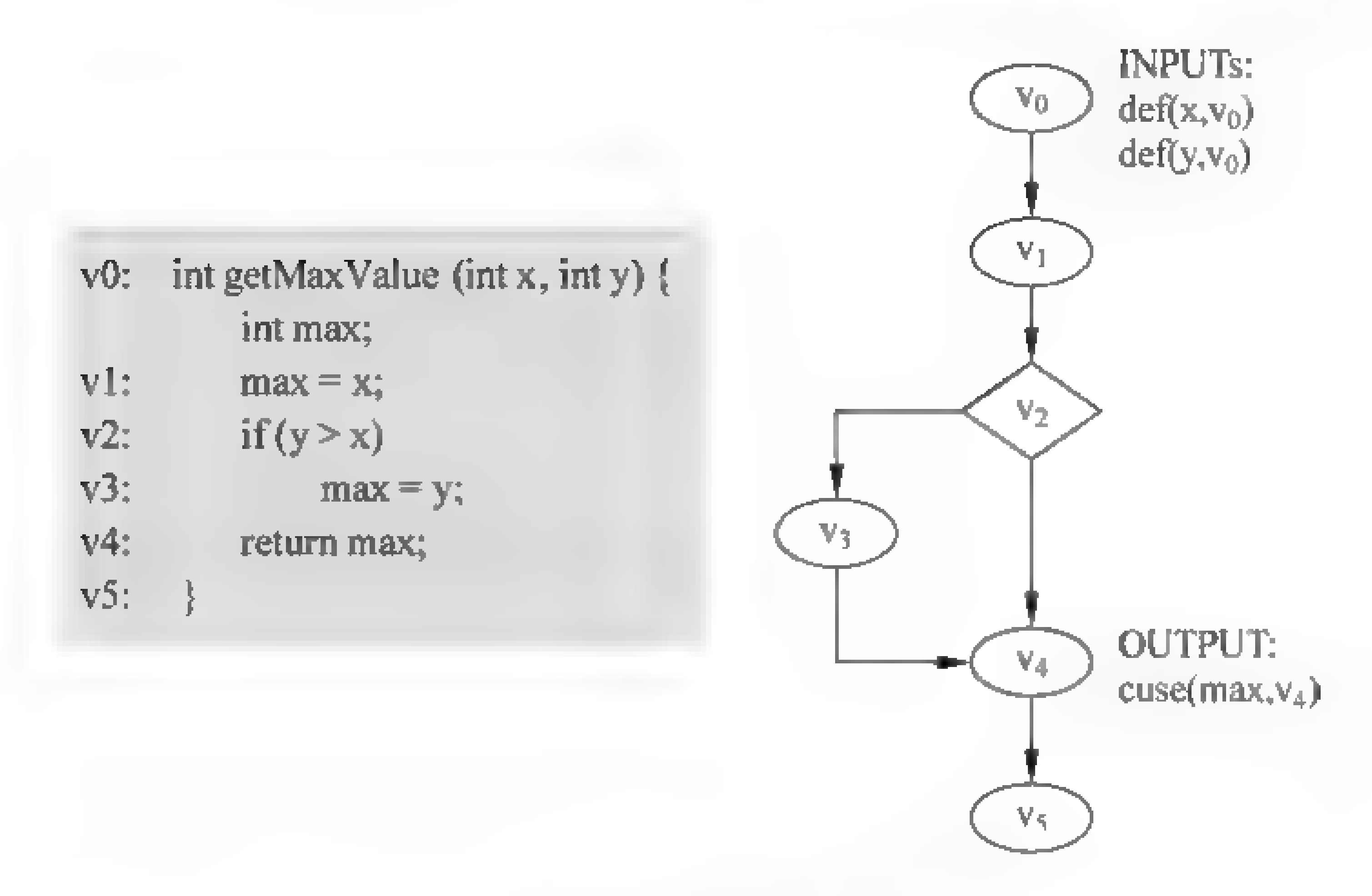


图 4-9 IO-df-链

为什么使用 ODC 而不是 du-对？如图 4-10 所示，变量 x 和 y 在节点 v_9 的使用受它们各自定义的影响；对于不同定义序列，这种影响可能是不一样的。如[def(x，v_2)，def(y，v_5)]和[def(x，v_3)，def(y，v_6)]是不同的 ODC，对应不同路径，可能产生不同测试效果。

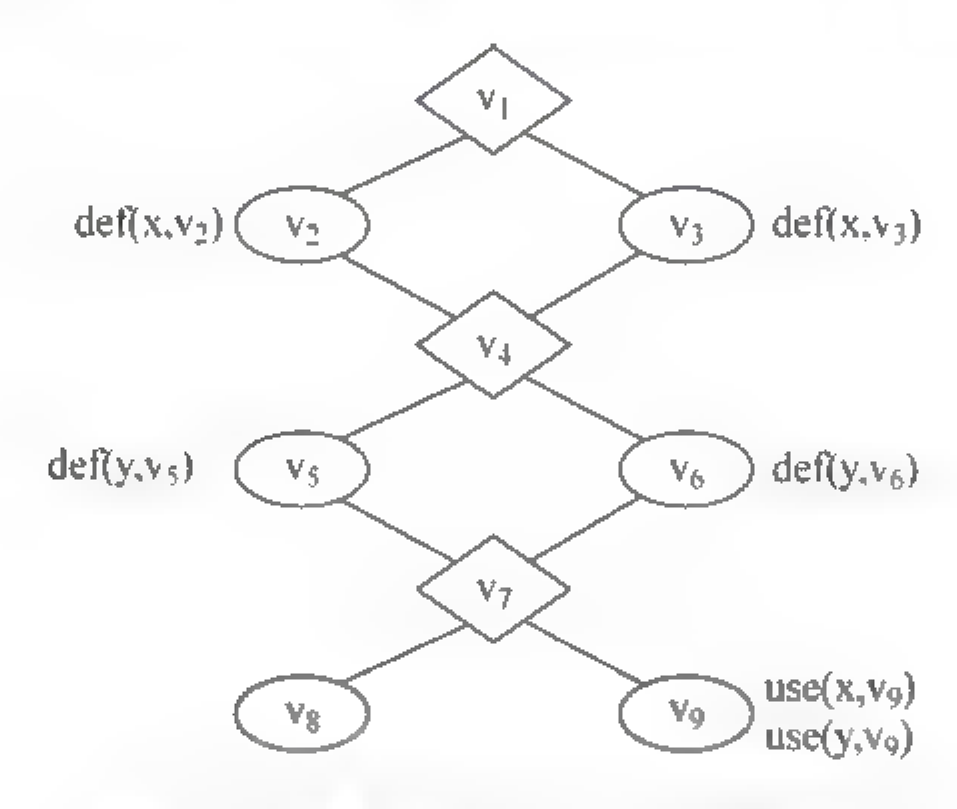


图 4-10 有序定义上下文（ODC）示例

一个测试组（Test Suite）被称作是满足有序定义上下文（ODC）覆盖标准：如果对于每一个节点和弧，该节点或弧的有序定义上下文被测试组的某一条路径覆盖。在图 4-5 中的程序图里，所有节点和弧的有序定义上下文为{[def(x，v_0)]，[def(x，v_0)，def(y，v_0)]，[def(y，v_0)]，[def(max，v_1)]}，分别被路径 $P_1=(v_0,v_1,v_2,v_3,v_4,v_5)$ 和 $P_2=(v_0,v_1,v_2,v_4,v_5)$ 覆盖，由路径 P_1 和 P_2 组成的测试组满足有序定义上下文（ODC）覆盖标准.

注意：Laski 和 Korel 的标准不能满足选择路径时要覆盖所有边和所有点的最低要求。

简单地总结一下数据流覆盖标准的优缺点。数据流覆盖标准的优点是所报告的路径与程序处理数据的方式直接相关；缺点是这类度量不包括判断覆盖，使用起来比较复杂。研

究人员已提出了许多种办法，所有这些办法增加了这类度量的复杂度。例如，有些办法区分在一个变量的使用是在计算中还是在判定中；是局部变量还是全局变量。和代码最优化的数据流分析一样，指针也带来问题。

4.4 其他覆盖标准

4.2节和4.3节分别介绍了控制流覆盖与数据流覆盖，它们都属于代码覆盖，也就是基于程序的覆盖分析。然而，覆盖可以延伸到黑盒测试。基于规格说明书，而不是基于代码确定覆盖是可能的。这当然要依靠所用规格说明的语言。对于较为规范的规格说明语言，比较容易提出一套覆盖标准。例如，为状态模型提出覆盖标准要比为用英语写的说明书提出覆盖标准更容易。本节简要介绍几种与程序无关的覆盖分析，包括数据域覆盖、统计或可靠性覆盖、风险覆盖、安全覆盖、基于需求的覆盖、基于状态模型的覆盖及基于错误的覆盖。

4.4.1 数据域覆盖

数据域覆盖(Data Domain Coverage)是基于等价类划分测试和边界测试。程序中的数据是无穷的，等价类划分测试能让我们把那些数据分离为几个部分。这么做，使得测试用例的数量测试运行时切实可行。那么如何知道我们所关心的用例是否覆盖了所有部分？数据域覆盖的主要步骤如下：

(1) 根据程序的逻辑判定把输入范围划分成次级范围。

(2) 检查那些子域在当前测试用例集里被覆盖的百分比。

数据域覆盖的优点是可以容易和迅速地管理无限数据，缺点是不能肯定划分是否足够完备。

4.4.2 统计或可靠性覆盖

统计或可靠性覆盖(Statistical or Reliability Coverage)是基于样本测试和随机测试的。由两次失败之间使用的平均数——MTTF(Mean Time To Failure，平均时间到失败)及公式MTTF=1/(1−可靠性)，通过计算可以得到可靠性。

4.4.3 风险覆盖

风险覆盖(Risk Coverage)基于风险分配(Risk Assignment)。风险是指一个潜在的问题，它是由事件、危险、威胁等情况发生的可能程度及发生后的不良后果来衡量的。风险部分与系统的可靠性密切相关。风险覆盖分析给出了有关可能导致危险的问题的一个系统的

可靠性。

4.4.4 安全覆盖

安全对于防御、医疗设备、航空航天、核和空间站应用是至关重要的。在取得测试结果之后，如何从结果中知道系统的安全程度？安全覆盖(Safety Coverage)能给出这样的度量。安全覆盖的主要步骤如下：

(1) 做 FTA、ETA 或 FMEA 分析以得到与安全有关的组件。

(2) 比较测试用例与那些组件。

因为安全覆盖是很重要的，所以需要高安全性。当检查安全覆盖时，我们需要确定它是100%安全的。

4.4.5 状态模型的覆盖标准

这种覆盖标准是基于状态模型(State Model)、状态机(State Machine)或 UML 的状态图(State Chart)，有以下几种度量：

- 状态覆盖是指测试用例集中被覆盖的状态数与给定状态模型里所有状态数的比率。
- 事件覆盖是指测试用例集中被覆盖的事件数与给定状态模型里所有事件数的比率。
- 转换覆盖是指测试用例集中被执行的转换数与给定状态模型里所有转换数的比率。
- 状态-事件覆盖是指测试用例集中被执行的状态-事件数与给定状态模型里所有状态数与所有事件数的乘积的比率。
- 路径覆盖是指测试用例集中从开始状态到结束状态之间的路径数与给定状态模型里从开始状态到结束状态之间的所有路径数的比率。
- 高风险的路径、转换和状态覆盖是指测试用例集中与风险有关的状态数、路径数以及转换数与给定状态模型里所有被识别的与风险有关的状态数、路径数以及转换数的比率。

基于状态模型覆盖标准的优点是，一旦一个系统的状态被正确地和完备地确定，我们几乎可以确信，测试用例覆盖了整个系统。即使需要增加测试用例，应该增加的测试用例也是容易计算出来的；缺点是不能肯定状态是完备的和正确的。那么如何能检查状态是完备的和正确的？这项工作是费时的。

4.4.6 覆盖标准有关问题、局限性

覆盖标准的选择是重要的，需要执行效益/成本分析。标准的数量是重要的，包括黑盒和白盒覆盖；白盒覆盖里控制流、数据流还有数据域覆盖。在测试上花费的时间是有限的。

软件经常会改变，因而早一些时候所保证的某些覆盖，但是软件改变了，也许就需要重新开发测试用例或重写脚本。

即使选择一系列的测试覆盖，有时也很难确定所需的确切的测试标准或很难执行测试覆盖。例如，假设想用等价类划分测试方法，并且在每个划分里要有至少3个测试用例，并且在边界值附近有4个测试用例。但是，这套标准仍不足够强大，因为一位测试工程师也许会有2个划分，但是一位好的测试工程师会有10个划分。他们两个都满足测试覆盖，但因为第二位测试工程师有更多的划分，他有更多的测试用例！提供覆盖并不能保证软件是高质量的。测试覆盖的评估可能是一项烦琐的任务，需要利用采样策略进行检查。

如果规格没有改变，那么同样的黑盒测试用例可以用于回归测试以保证相同的覆盖。不幸的是，如果软件改变了，那么无论是规格或代码变动而引起的变化，经常需要更新相应的白盒测试用例或是重新开发。如果软件改变了，则经常需要更新相应的覆盖。

Poston 描述了这样常见的情况，即当接近最后期限时，工程师倾向于说有些错误是不重要的或很少发生的因而没有必要修改它。这的确是一个危险做法，但是它确实会发生，特别是当最后期限是接近时。当最后期限接近时，变动率增加。我们所观察到是变动率在最后期限达到高峰。

4.4.7 实际应用的建议

现代发展强调的是速度，例如基于测试的发展有极限编程、敏捷过程和 E2E 测试等。在这种情况下，建议应该在可交付使用前中期时执行覆盖测试，即强调黑盒测试覆盖与灰盒测试覆盖(例如灰盒路径)；否则当软件迅速改变时，维护一些测试覆盖简直是不可行的。

由于每种覆盖代表一种具体的测试技术，以及对于软件和关注的问题应用这种测试技术进行应该执行的程度范围。因此在提出建议之前仔细地分析问题是根本的要求。

一般来说，要点是要仔细审查情形与场合。例如，实时系统可能经常需要控制流覆盖，数据应用需要数据，可能需要数据流覆盖。数据库事务系统经常要求处理并发事务，因此它也需要控制流覆盖。特殊系统也需要相应的特殊覆盖，例如时间分析覆盖、并发分析覆盖。如果系统在维护或开发中，规定回归测试覆盖和影响分析覆盖是重要的。效益/成本分析也是重要的，选择一个覆盖目标达到100%有时是不可能的，但应该避免制定的目标低于80%。

覆盖测试是验证软件功能结构正确性以及查找问题的非常重要的方法和手段。在什么情况下代码覆盖是适合的测试技术呢？在黑盒测试中，只使用规格说明书去执行测试。在白盒测试中，需要知道某些变量在一个指定范围之内被设计取值。代码覆盖是一种白盒测试方法，因为代码覆盖要求代码的知识并且浏览代码。

4.5 总结

软件测试覆盖分析是一种可以凭经验确定软件质量的方法。在测试计划阶段，测试者确定用何种测试覆盖分析及相应的覆盖率；在测试执行阶段，将根据既定的覆盖率来检查是否进行了足够的测试。

面向白盒测试技术的覆盖分析主要是代码覆盖分析。代码覆盖分析是要针对一个待测程序和一个覆盖标准，产生一个测试组（一组满足该覆盖标准路径的有限集）。代码覆盖分析主要有两种类型：控制流覆盖与数据流覆盖。前者是选择一组实体以满足一定的覆盖标准，比如语句覆盖、判定覆盖、条件覆盖、多条件覆盖、条件判定组合覆盖、修正条件/判定覆盖及路径覆盖；后者选择一组满足变量的定义与引用间的某种关联关系实体，选择的标准主要有 Rapps 和 Weyuker 的标准、Ntafos 的标准、Ural 的标准及 Laski 和 Korel 的标准。

面向黑盒测试技术的覆盖分析主要是基于规格说明书进行的，覆盖标准主要包括数据域覆盖、统计或可靠性覆盖、风险覆盖、安全覆盖、基于需求的覆盖、基于状态模型的覆盖及基于错误的覆盖。

由于每种覆盖代表一种具体的测试技术，以及对于软件和关注的问题应用这种测试技术进行应该执行的程度范围，因此在实际应用中提出建议之前需要仔细地审查情形与应用场合。通常建议在可交付使用前中期时执行覆盖测试，即强调黑盒测试覆盖与灰盒测试覆盖（例如灰盒路径）。

4.6 参考文献

[1] Joan C. Miller, Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 1963; 6(2): pp. 58～63

[2] Ntafos, Simeon. A Comparison of Some Structural Testing Strategies. *IEEE Trans. Software Eng.*, 1988; Vol. 14, No. 6, pp. 868～874

[3] Marc Roper. *Software Testing*. London: McGraw-Hill Book Company, 1994

[4] John Joseph Chilenski, Steven P. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal*. September 1994, Vol. 9, No. 5, pp. 193～200

[5] Rapps, S., Weyuker, E. J.. Selecting Software Test Data Using Data Flow Information, Software Engineering. *IEEE Transactions*. 1985: 367～375

[6] Hasan Ural. Test sequence selection based on static data flow analysis. *Computer Communications*. 1987, 10(5): pp. 234～242

[7] Laski, J. W., Korel, B.. A Data Flow Oriented Program Testing Strategy, Software Engineering. *IEEE Transactions*. 1983: pp. 347～354

4.7 思考与练习

1. 试用自己的话描述什么是软件测试覆盖分析及其作用。

2. 试比较各种控制流覆盖标准的优缺点。

3. 阅读下面这段程序，试用语句覆盖、判定覆盖、条件覆盖及路径覆盖分析技术对其进行测试。请列出所用的测试用例，并分析每种方法的覆盖率。

```
READ X,Y;
IF Y<0 THEN
POW :=-Y;
ELSE
POW :=Y;
Z :=1;
WHILE(POW! =0)
{Z :=Z * X;
POW :=POW-1; }
IF Y<0 THEN
Z :=1/Z;
ANSWER :=Z+1;
PRINT ANSWER;
```

4. 为什么需要使用数据流覆盖分析？

5. 试分析路径选择的一组覆盖标准之间的涵盖关系。请用 all-du-paths 标准为第 3 题中的程序设计测试用例。

6. 试结合一个你参与或听说过的实际的软件开发过程，思考如何应用软件测试覆盖分析技术。

4.8 进一步阅读

Andrew Glover, In Pursuit of Code Quality: Don't be Fooled by the Coverage Report, http://www 128. ibm. com/developerworks/java/library/j cq01316/? ca dnw 704, 2006

BCS SIGIST (British Computer Society Specialist Interest Group in Software Testing), Standard for Software Component Testing, 2001

L. A. Clarke, A. Podgurski, D. J. Richardson. A Formal Evaluation of Data Flow Path Selection Criteria. IEEE Transactions on Software Engineering. 1989: Vol. 15, No. 11, pp. 1318～1332

Lasse Koskela. Introduction to Code Coverage. Accenture Technology Solutions. 2004

J. R. Horgan. S. London. Data Flow Coverage and the C Language. New York: International Symposium on Software Testing and Analysis,1991: pp. 87～97

S. J. Zeil. Selectivity of Data-Flow and Control-Flow Path Criteria. Canada: ACM SIGSOFT/IEEE Second Workshop Software Testing, Verification, and Analysis, 1988: pp. 216～222

Steve Cornett. Code Coverage Analysis. http://www. bullseye. com/coverage. html

W. E. Howden. Confidence-Based Reliability And Statistical Coverage Estimation. Eighth International Symposium on Software Reliability Engineering (ISSRE '97). 1997: p. 283

McCabe, Tom. A Software Complexity Measure. IEEE Trans. Software Eng. , 1976: Vol. 2, No. 6, pp. 308～320

Sandra Rapps, Elaine J. Weyuker, Data Flow Analysis Techniques for Test Data Selection. Los Alamitos: Proceedings of the 6th International Conference on Software Engineering. 1982: pp. 272～278

第5章

单元测试与集成测试

分阶段测试是一种基本的测试策略。最初，测试着重于每一个单独的模块，以确保每个模块都能正确执行，因此称为单元测试。单元测试大量使用白盒测试技术。检查每一个控制结构的分支以确保完全覆盖和最大可能的错误检查；接下来，模块必须装配或集成在一起形成完整的软件包，集成测试解决的是功能验证与程序构造的双重问题，在集成过程中使用最多的是黑盒测试用例设计技术。当然，为了保证覆盖一些大的分支，也会在一定范围内使用白盒测试技术；在软件集成（构造）完成之后，一系列高级测试就开始了。最后的高级测试步骤已经超越了软件工程的边界，而属于范围更广的计算机系统工程的一部分。软件一旦经过验证之后，就必须和其他的系统元素（比如硬件、人员、数据库）结合在一起。系统测试要验证所有的元素能正常地啮合在一起，从而完成整个系统的功能/性能目标。确认标准（在需求分析阶段就已经确定了的）必须进行测试，确认测试最后确保了软件符合所有功能的、行为的和性能的需求，在确认过程中，只使用黑盒测试技术。

本章只介绍单元测试与集成测试概念、目标及过程。

快速浏览：

什么是单元测试、集成测试？ 单元测试（Unit Testing）是对最小的软件设计单元（模块或源程序单元）的验证工作。集成测试（Integration Testing）把单独的软件模块结合在一起，作为一个群接受测试。

由谁来负责单元测试、集成测试？ 单元测试一般由软件开发人员进行。一般由有经验的软件测试人员和软件开发人员共同完成集成测试的计划和执行。

为什么单元测试、集成测试如此重要？ 单元测试有助于消除单元本身的一些不确定性，它可以应用于自底向上的测试中。这样先测试程序的一部分，之后再将部分组合起来测试整体，使得集成测试更加容易。集成测试的目的在于：基于主要的单元集合，来验证功能、性能和可靠性的需求。

单元测试、集成测试步骤各是什么？ 单元测试一般使用白盒测试技术，也可以使用黑盒

测试技术，其测试步骤与白盒测试或黑盒测试步骤相同。集成测试步骤与集成测试策略有关。

有哪些工件形成？ 在一些情况下，会生成测试计划和测试用例。在每一种情况下，要将测试结果存档以便将来软件维护时使用。

如何确保准确完成了任务？ 尽管永远不能保证已经执行了所要求的每一个单元测试、集成测试，但能肯定测试已经发现了错误（并且已修正了这些错误）。另外，如果已经制定了一个测试计划，那么可以检查以保证所有计划测试已被完成。

5.1 单元测试

单元测试是对最小的软件设计单元（模块或源程序单元）的验证工作。从更专业的角度看，应该把一个单元理解成一个应用程序中最小的可测部分。在面向过程的设计（Procedural Design）中，一个单元可能是单独的程序、函数、过程。网络应用的设计（Web-baced application design）中，一个单元可能是单独的网页以及菜单等。而在面向对象的设计（Object Oriented Design）中，最小单元永远是类，可能是基/父类、抽象类或派生/子类。单元测试使用构件级别的设计规格说明书作为指南，对重要的控制路径进行测试以发现模块内的错误。单元测试把重点放到内部处理逻辑和构件边界内的数据结构。这种测试可以对多个构件并行进行。通常情况下，单元测试是由开发者执行测试而不是由最终用户执行测试，主要使用白盒测试技术，并辅助使用黑盒测试技术，如边界值分析法。执行测试和发现的错误的相对复杂度取决于所设立的单元测试的限制范围。

注意，把 Unit Testing 和 Unit Test 翻译成中文时，都译成“单元测试”。要注意它们的区别：前者表示单元测试的测试类型、一般过程等；而后者对于某特定单元的一次测试。由于从中文的“单元测试”看不出这种区别，所以要通过上下文去理解。

5.1.1 单元测试考虑事项

单元测试对构件的 5 方面进行测试：模块或构件接口、局部数据结构、边界条件、独立路径和处理错误的路径。

1. 模块或构件接口

对模块接口的测试要保证在测试时进出程序单元的数据流是正确的，包括接口名称，传入参数的个数、类型、顺序等是否与模块接口匹配；模块输出或返回值或类型是否正确。对穿越模块接口的数据流的测试需要在任何其他测试开始之前进行，如果数据不能正确地输入和输出的话，所有的其他测试都是没有实际意义的。

2. 局部数据结构

对局部数据结构的检查应保证临时存储的数据在算法执行的整个过程中都能维持其完整性，并且通过单元测试确认执行过程中局部数据结构对于全局数据的局部影响。

3. 边界条件

对边界条件的测试以保证模块在所限定或约束处理的条件边界上能够正确执行。边界测试是单元测试任务的一项重要步骤。软件通常是在边界情况下出现故障的，这就是说，错误往往出现在一个 n 元数组的第 n 个元素被处理的时候，或者当一个 i 次循环的第 i 次调用，或者当允许的最大或最小数值出现的时候。使用刚好小于、等于和刚好大于最大值和最小值的数据结构、控制流、数值来作为测试用例就很有可能发现错误。边界条件的测试是利用黑盒测试技术中的边界值分析法。

4. 独立路径

在控制结构中的所有独立路径（基本路径）都要走遍，以保证一个模块中的所有语句都能执行至少一次。在单元测试过程中，对执行路径的选择性测试是最主要的任务。测试用例应当能够发现由于错误计算、不正确的比较或者不正常的控制流而产生的错误。基本路径测试和循环测试是发现更多的路径错误的一种有效技术。

1）计算中常见的错误

（1）误解的或者不正确的算术优先级。

（2）混合模式的操作。

（3）不正确的初始化。

（4）精度不够。

（5）表达式中不正确符号表示。

比较和控制流是紧密地耦合在一起的（也就是说，控制流的转移是在比较之后发生的）。

2）测试用例应当发现的错误

（1）不同数据类型的比较。

（2）不正确的逻辑操作或优先级。

（3）应该相等的地方由于精度的错误而不能相等。

（4）不正确的比较或者变量。

（5）不正常的或者不存在的循环终止。

（6）当遇到分支循环的时候不能退出。

（7）不适当地修改循环变量。

5. 处理错误的路径

最后要对所有处理错误的路径进行测试。好的设计要求错误条件是可以预料的，而且

当错误真的发生的时候，错误处理路径被建立，以重定向或者干脆终止处理。Yourdon[2]把这种方法叫做反调试(Antidebugging)。不幸的是，存在一种倾向，就是把错误处理过程加到软件中去，但从不进行测试。有一个现实生活中的故事可以说明这个问题："一个交互式设计系统按照合同进行开发。在一个事务处理模块中，开发人员将错误处理信息"错误！你不可能到达这里！"加到调用各种控制流分支的一系列条件测试之后。这个"错误信息"在培训过程中被一个用户发现了！"

在错误处理部分应当考虑的潜在错误有如下几种情况：

- 对错误描述不易理解。
- 所报的错误与真正遇到的错误不一致。
- 在错误处理之前错误条件先引起系统功能之间的干扰，造成系统异常。
- 异常条件处理不正确。
- 错误描述没有提供足够的信息来帮助确定错误发生的位置。

5.1.2 单元测试规程

单元测试通常被看作附属于编码步骤。在对源代码级的代码进行开发、复审和语法正确性验证之后，单元测试用例设计就开始了。对设计信息的复审可能能够为前面讨论过的每一类错误的测试用例提供指导，每一个测试用例都应当和一系列的预期结果联系在一起。

因为一个模块本身不是一个独立的(Stand-alone)程序，所以必须为每个单元测试开发驱动器(Driver)或/和程序桩(Stub)。在绝大多数应用中，一个驱动器只是一个"主程序"，负责接收测试数据，并把数据传送给(要测试的)模块，然后打印相关结果。子程序桩的功能是替代那些隶属于被测模块(被调用)的模块。一个子程序桩或"空子程序"使用被调子模块的接口，可能要做一些最少量的数据操作，并打印入口处验证的信息，然后把控制返回给被测模块。在面向对象的程序里，模仿对象(Mock Object)技术取代程序桩。模仿对象是以一种可控方式来模拟真实对象行为的仿真对象。我们将在第6章详细介绍这种技术。

驱动器和程序桩都是额外的开销，也就是说，两种都属于必须开发但又不和最终软件一起交付的软件。如果驱动器和程序桩很简单，那么额外开销相对来说是很低的。当一个模块被设计为高内聚松耦合时，单元测试是很简单的。当一个模块只表示一个函数时，测试用例的数量就会降低，而且错误也就更容易被预测和发现。不幸的是，许多模块使用"简单"的额外软件是不能进行足够的单元测试的。在这些情况下，完整的测试要推迟到集成测试步骤时再完成。

单元测试通常是被自动执行的，但也可能手工进行。有关软件单元测试的IEEE标准[3]并没有规定是用自动的还是手工的方法进行单元测试。手工方法可以按照指示文档一步一步地进行。然而，单元测试的目标是要隔离一个单元并验证其正确性，自动化方法能有

效地达到这一目标，使单元测试取得应有效果。使用自动化方法，为完全实现隔离效果，进行单元测试的代码体是在其自然环境以外的框架内运行的，也就是说，在产品以外或被原始创建时调用环境以外。以隔离方式进行测试有暴露不必要依赖的好处，这种依赖是在被测代码与产品中其他单元或数据空间之间。如果需要，这些依赖可以通过重构或重新设计来改进。

使用单元测试，开发人员把一些标准写成代码放入单元测试里来验证被测单元的正确性。在执行单元测试过程中，框架把所有失败的测试用例记入日志。许多框架还自动在一个汇总表里标识并报告这些没有通过的测试用例，根据问题的严重程度，框架可能中止后续的测试。

因此，单元测试成为一种动力，它驱使程序员创建松耦合、高内聚的代码体，这种实践有助于形成健康的软件开发的习惯。设计模式、单元测试和重构常结合使用以便形成最理想的解决方案。第6章将继续介绍有关这方面的内容。

5.1.3　单元测试局限性

单元测试不能捕获程序中的每一个错误。根据定义，单元测试只测试单元自身的功能。因此它不捕获集成错误、性能问题或其他任何系统范围的问题。另外，要预料现实中被测程序可能接收到的输入的所有特殊情况是很困难的。对于任何非平凡(Nontrivial)的软件块，要测试其所有的输入组合是不现实的。如软件测试的其他所有形式，单元测试只能展示错误的存在，而不能展示错误不存在。

要从单元测试中获得预期效果，软件开发的整个过程中需要严格的科学专业素养。不仅要认真维护已经执行的测试的记录，还要记录源程序或软件中其他单元所发生的变化。使用所谓"版本控制系统"是基本的：如果单元的后来版本没有通过前面测试已通过的某一个用例，那么版本控制系统可以提供自上次测试后对于单元所施与的变化。这有助于发现在哪次、因什么变化使得测试用例没能通过。

单元测试只有和其他软件测试活动，如集成测试、系统测试结合起来使用才能有效。5.2节将具体介绍集成测试。

5.2　集成测试

集成测试(Integration Testing)，有时也称作集成与测试(I&T)，这是软件测试的一个阶段，在这个阶段单独的软件模块被结合在一起，作为一个群接受测试。什么时候进行集成测试？在如下3种情况下进行需要进行集成测试：

(1) 由若干单元或模块组成一个构件。

(2) 由若干构件组成为一个工件。

(3) 由若干工件组成为一个系统。

集成测试被定义为在单元测试与系统测试之间级别的测试。在所有的模块都已经完成单元测试之后，有人或许会问这样一个似乎很合理的问题："如果它们每一个都能单独工作得很好，那么为什么要怀疑把它们放在一起就不能正常工作呢？"当然，这个问题就在于"把它们放在一起"——即接口连接问题。数据可能在通过接口的时候丢失；在连接时一个模块可能对另外一个模块产生无法预料的副作用；当子函数被联到一起的时候，可能达不到期望的功能；在单个模块中可以接受的不精确性在联接起来之后可能会扩大到无法接受的程度；全局数据结构可能也会存在问题。诸如此类的问题还可以列举很多。

集成测试被看作是一种系统化技术，用来构造程序并实施测试以发现与接口连接有关的错误，它的目标是把通过了单元测试的模块拿来，构造一个在设计中所描述的程序结构。有两种集成测试策略：瞬时集成测试(Instantaneous Integration Testing)和增量集成测试(Incremental Integration Testing)。

- **瞬时集成测试**有时候被称为"大爆炸(Big Bang)"的方法，属于非增量集成测试(Non-incremental Integration Testing)的范畴。由 Myers 在 1979 年定义的一种方法，当所有被隔离的构件都通过了测试，就把它们组合成一个最终系统，并观察它是否运转正常。这种方法的结果通常是混乱不堪！会遇到许许多多的错误，错误的修正也是非常困难的，因为在整个程序的庞大区域中想要分离出一个错误是很复杂的。一旦这些错误被修正之后，马上就会有新的错误出现，这个过程会继续下去，而且看上去似乎是个无限循环的。
- **增量集成测试**是大爆炸方法的对立面。程序先分成小的部分进行构造和测试，这个时候错误比较容易分离和修正；对接口也更容易进行彻底的测试；而且也可以应用一种系统化的测试方法。增量集成测试会有额外的开销，但会大大减少发现和改正错误的时间，最佳的增量方法本质上取决于各个项目和不同利弊选择的考虑。

许多程序员在开发小程序的时候都会用到瞬时集成测试技术，但这种技术对大型程序不太适用。事实上，瞬时集成方法有这样几个缺点：

(1) 对独立组件测试需要驱动程序和树桩程序的支持。

(2) 由于所有组件都是一次性地结合在一起的，所以很难找出错误的原因。

(3) 不容易辨别接口错误和其他类型的错误。一般情况下，不推荐将瞬时集成用于任意系统，而是推荐使用增量集成策略。

在 5.2.1 节至 5.2.3 节中，将介绍 3 种增量集成测试方法：自顶向下集成、自底向上集成和混合式集成。5.2.4 节将介绍端到端(E2E)集成测试。通常的集成测试比较注重接口连接测试，而 E2E 集成测试是专注于从终端用户角度的系统功能的测试，是在完成通常的集成测试后进行的。

5.2.1 自顶向下集成

自顶向下集成是一种构造程序结构的增量实现方法[1]。模块集成的顺序是首先集成主控模块(主程序),然后按照控制层次结构向下进行集成。隶属于(和间接隶属于)主控模块的模块按照深度优先或者广度优先的方式集成到整个结构中去。

如图 5-1 所示,深度优先集成是集成结构中的某一个主控路径上的所有模块。主控路径的选择是有些任意的,它依赖于应用程序的特性,例如,选择图中最左边的路径,模块 M_1、M_2 和 M_5,将会首先进行集成,然后是 M_8 或者是(如果对 M_2 的适当的功能是必要的) M_6,然后开始构造中间的和右边的控制路径。广度优先的集成是沿着水平的方向,把每一层中所有直接隶属于上一层模块的模块集成起来,从图 5-1 中来说,模块 M_2、M_3 和 M_4 首先进行集成,然后是下一层的 M_5 和 M_6,然后继续。

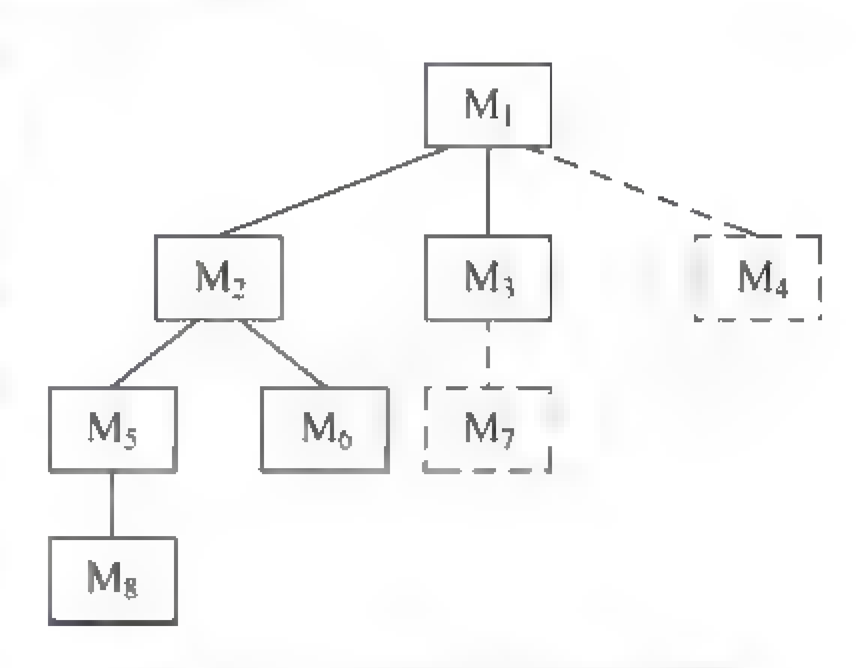


图 5-1 自顶向下的集成示例

自顶向下测试采用深度优先方法时,每一模块在测试中逐层由真实代码替代程序桩,使该模块得到不断的详尽的测试。自顶向下测试采用广度优先方法时,应用程序中处于同一控制层上的模块在进行测试时得到精化。在现实中一般是结合使用这两种技术进行测试。在初始阶段所有的模块可能只是提供部分功能,这可以用宽度优先技术进行测试,过了一段时间后,模块被越来越精化,模块的功能也越来越全,这时候就可以对一个模块进行深度优先测试而同时所有的模块进行宽度优先测试。

自顶向下集成策略主要由下列 5 个步骤来完成:

(1) 主控模块作为测试驱动器,所有的程序桩由直接隶属于主控模块的各模块替换。

(2) 根据集成的实现方法(如深度或广度优先),子模块的程序桩依次被替换为实际模块。

(3) 在每一个模块集成的时候都要进行测试。

(4) 在完成了每一次测试之后,又一个程序桩被实际模块替换。

(5) 可以用回归测试(第 7 章)来保证没有引进新的错误。

整个过程回到第(2)步循环进行,直至整个系统完成构造。

自顶向下的集成策略在测试过程的早期就会验证主要的控制和决策点。在一个好的程序结构中,决策往往发生在层次结构中的高层,用自顶向下的集成测试决策问题首先会被发现。如果主控制的确存在问题,那么尽早发现它是很重要的。如果选择了深度优先集成,那么软件的某个完整的功能会被实现和证明,例如,考虑一个典型的事务性结构。此结构中,一系列复杂的交互式输入要通过一条输入路径进行请求、获得和验证,这条输入路径就可以

用自顶向下的方式来进行集成。早期的对功能性的验证对开发人员和客户来说都会增加信心。

自顶向下的策略听起来似乎不是很复杂，但是在实践过程中，可能会出现逻辑上的问题。最普遍的问题出现在当高层测试需要首先对较低层次进行足够测试后才能完成的时候。在自顶向下的测试开始的时候，程序桩代替了低层的模块，因此，在程序结构中就不会有重要的数据向上传递，测试者只有下面的 3 种选择：

(1) 把测试推迟到程序桩被换成实际的模块之后再进行。

(2) 开发能够实现有限功能的程序桩，用来模拟实际模块。

(3) 从层次结构的最底部向上来对软件进行集成。

第一种实现方法(把测试推迟到程序桩被换成实际的模块之后再进行)使我们失去了对特定测试之间通信的控制和对特定模块组合的控制，这样可能导致在确定错误发生原因时的困难性，并且会破坏自顶向下方法具有的高度受限的本质；第二种方法是可行的，但是会导致很大的额外开销，因为程序桩会变得越来越复杂；第三种方法，也就是自底向上的测试，将在 5.2.2 节加以讨论。

自顶向下集成的优点如下：

(1) 早期对高层行为进行确认。

(2) 至多只需一个驱动程序。

(3) 每步可以只加一个模块。

(4) 支持深度优先和宽度优先。

自顶向下集成的缺点如下：

(1) 对低层行为的确认比较晚。

(2) 对缺少的元素需要编写树桩程序。

(3) 测试用例的输入和输出可能很难明确表示。

5.2.2 自底向上集成

自底向上的测试是从原子模块(也就是在程序结构的最低层的模块)开始来进行构造和测试的[4]。每个模块由测试装置(Test Harness)进行测试。一旦各个独立的模块测试完毕，就把它们组合起来形成一组模块，这组模块称为造件(Build)。一组造件再由第二个测试装置进行测试。这个过程将一直进行直到造件中包括整个应用系统。因为模块是自底向上集成的，在进行时要求所有隶属于某个给定层次的模块总是存在的，而且也不再有使用程序桩的必要。

自底向上的集成策略可以使用下列步骤来实现：

(1) 低层模块组合成能够实现软件特定子功能的造件，有时也称为簇(Cluster)。

(2) 写一个测试装置(一个供测试用的控制程序)来协调测试用例的输入输出。

(3) 对簇进行测试。

(4) 撤去测试装置,沿着程序结构的层次向上对造件进行组合。

这样的集成遵循如图5 2中所说明的模式,首先把所有的模块聚集成3个簇:簇1、簇2和簇3,然后用对每一个簇使用驱动器(图中的虚线框的块)进行测试,在簇1和簇2中的模块隶属于M_a,把驱动器D_1和D_2去掉,然后把这两个簇和M_a直接连在一起。类似地,驱动器D_3也在模块M_b集成之前去掉。M_a和M_b最后都要和模块M_c一起进行集成。

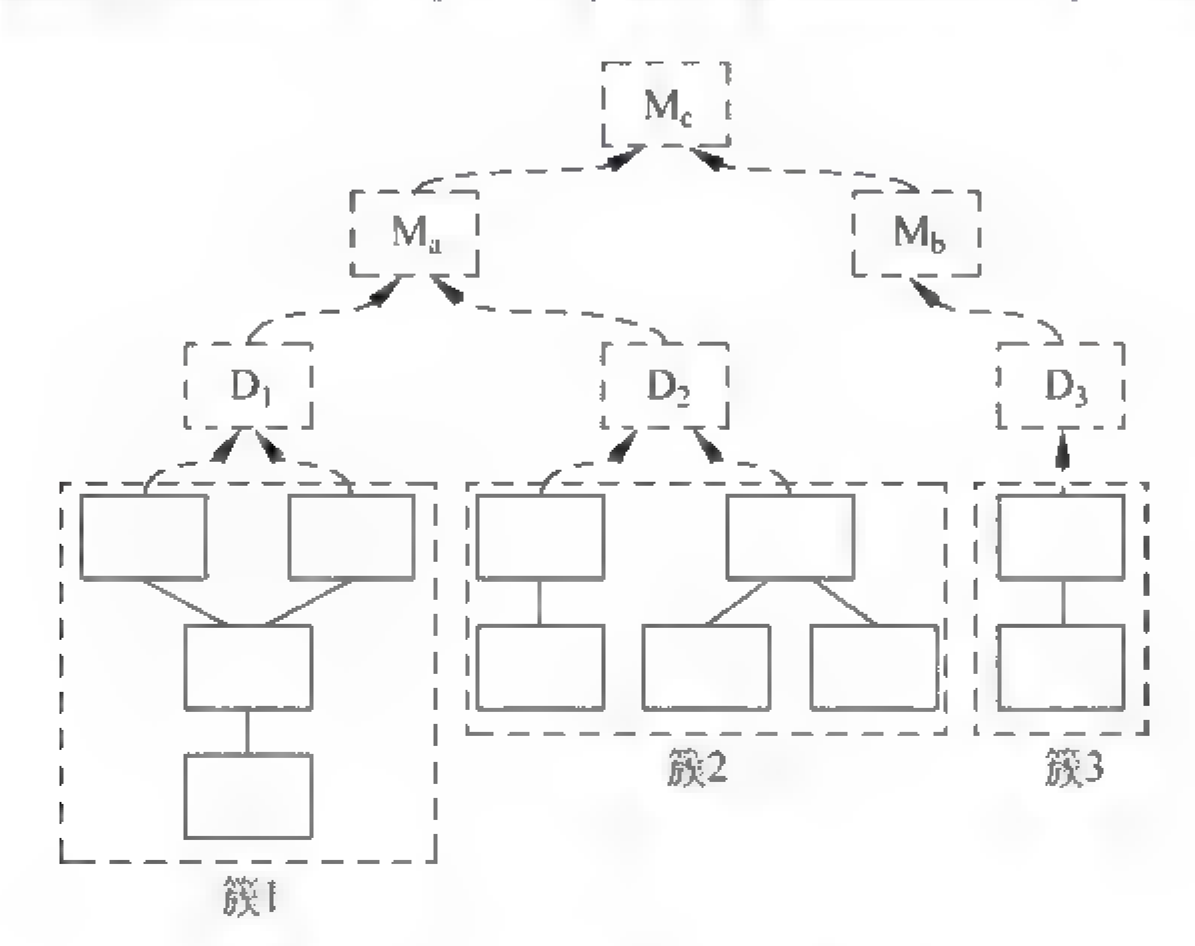


图5-2 自底向上的集成示例

当测试在向上进行的过程中,对单独的测试驱动器的需求减少了,事实上,如果程序结构的最上两层是自顶向下集成的,那么所需的驱动数目就会明显减少,对簇的集成从而会变得非常简单。

自底向上集成的优点如下:

(1) 早期对底层行为进行确认。

(2) 不需要写程序桩。

(3) 对一些子树而言比较容易明确表示输入,比较容易解释对其他的输出。

自底向上集成的缺点如下:

(1) 推迟对高层行为的确认。

(2) 需要驱动程序。

(3) 当组合子树的时候,有许多元素要进行集成。

5.2.3 混合式集成

在实际中测试通常是结合了自顶向下和自底向上这两种方法,称作混合式集成测试(Mixed Testing),也称作三明治式集成测试(Sandwich Testing)。在由几个小组一起参与开发

的大的软件项目中，或者一个小项目但不同模块是由不同的人进行构建的情况下，小组或个人可以对自己开发的模块采用自底向上测试，然后再由集成小组进行自顶向下测试。

混合式集成策略可以使用下列步骤来实现：

(1) 用程序桩独立地测试用户界面。

(2) 用驱动程序测试最低层功能模块。

(3) 当集成整个系统时，只有中间层是要进行测试的对象集。

如图 5-3 所示，使用程序桩 S_2、S_3 和 S_4 对用户界面 M_1 进行测试；使用驱动程序 D_5 和 D_6 对最低层功能模块 M_7、M_8 和 M_9 进行测试。当整个系统集成时，将程序桩 S_2、S_3 和 S_4 换成中间层模块 M_2、M_3 和 M_4；驱动程序 D_5 和 D_6 对换成中间层模块 M_5 和 M_6，从而对中间层的功能模块进行测试。

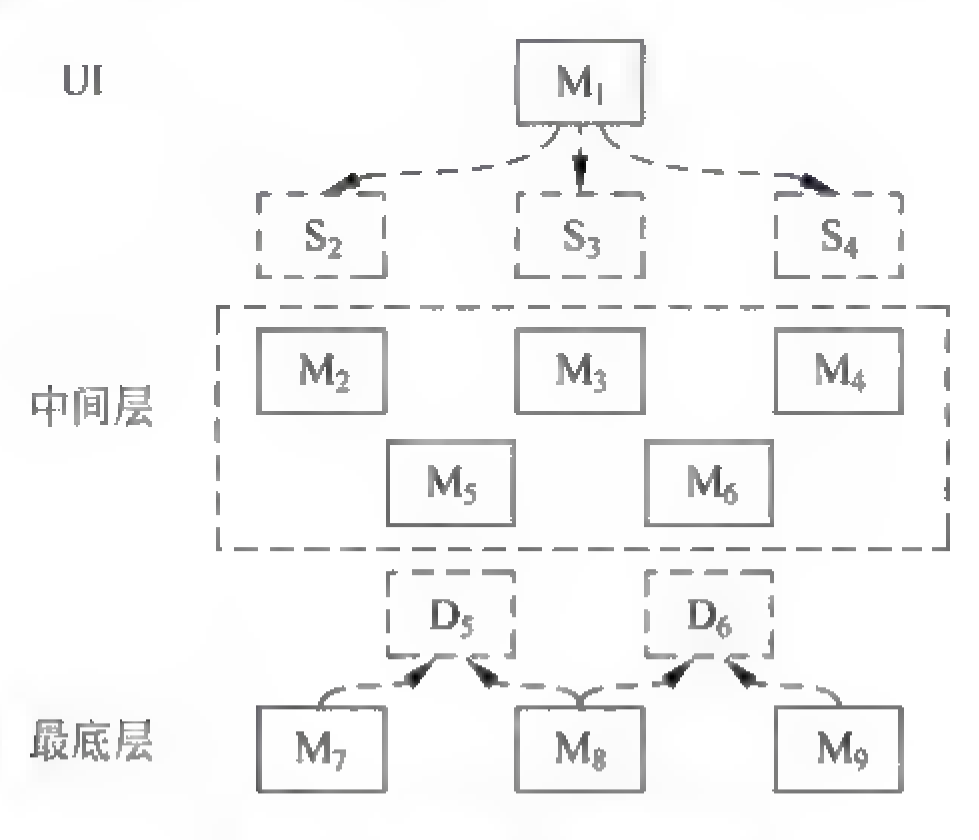


图 5-3 混合式集成示例

混合式集成测试是测试大型系统一种策略，属于增量测试技术。对于许多系统，混合式集成测试都是行之有效的，因为它综合了自顶向下集成策略和自底向上集成策略的优点。表 5-1 从 6 个方面对 3 种增量集成测试策略进行了对比。

表 5-1 3 种增量测试策略的比较

	自顶向下	自底向上	混合式
形成基本可工作程序所需时间	早	晚	早
是否需要构件驱动器	否	是	是
是否需要程序桩	是	否	是
集成开始时可否平行工作	低	中等	中等
测试特殊路径的能力	难	易	中等
计划和控制顺序的能力	难	易	难

上面分别介绍了 3 种增量集成测试策略，并进行了对比。在实际应用时，可以结合其他考虑因素，灵活使用增量集成测试策略(称之为混合增量集成方法)，如风险驱动、进度驱动

以及功能或线程驱动。

- **风险驱动**：从最关键或最复杂的模块开始进行集成，逐步加入它们调用或被调用的模块。
- **进度驱动**：一旦模块就绪，比如，以某种方式可以获得或编码完成，就马上进行集成。
- **功能或线程驱动**：选择跟某一个功能或线程有关的模块进行集成，逐步加入其他功能或线程。

5.2.4 端到端集成测试

传统的集成测试方法注重测试接口连接。与传统的集成测试不同，端到端集成测试完全从最终用户的角度出发，强调对系统或应用程序进行端到端的功能测试(Functional Testing)。这一测试过程用于验证由一组相互连接的系统形成的集成系统是否正常运行，其中每个被连接的系统现在都是集成系统的一个子系统。端到端集成测试(E2E Integration Testing)一般是面向大型系统的。端到端集成测试假设子系统的模块(或单元)测试和集成测试都已被执行并得到认可，但可能依然存在未被观察到的错误，其中集成测试可能包括多层次的集成测试[4]。

端到端集成测试独立于任何一个开发过程，可以在开发生命周期的早期开始，在此过程中测试需求分析可以和应用开发并行进行。端到端集成测试提供以下功能：

- 帮助生成测试用例，改进软件项目的生产率。
- 支持风险分析，通过确定风险领域从而进行全面的测试。
- 支持变更管理，从而使回归测试和涟漪效应测试可以被正确和有效的执行。
- 支持数据质量评估，从而使决策者能够定量地、更客观地评估测试成果。
- 支持远程项目管理和分布式协作使工程师和项目经理可以通过网络一起工作。

下面介绍终端到终端(E2E)的测试过程的各个阶段：

- 测试计划：确定主要任务及与其相关的进度安排和资源。
- 测试设计：开发测试规范、测试场景、测试用例和测试进度表。
- 测试执行：执行测试用例和文档结果。
- 测试结果分析：分析测试结果覆盖率，评估测试并确定过失。
- 重新测试和回归测试：在改进后的系统上进行附加测试。

1. E2E 集成测试计划

E2E 集成测试计划的核心是确定集成系统的范围，包括系统构架和功能；确定处理方法和工具以便于完成集成测试，并在实际测试之前制定结果评估标准。在一个 E2E 测试计划中需要考虑一些重要的因素包括：主要的目标、测试范围、系统功能和非功能需求、测试

环境、测试自动控制、测试结果分析计划、测试重用计划、系统支持计划、活动时间表和退出标准。建立测试工作小组，最好的用于确定E2E测试所需信息的方法是通过形成一个测试工作组来建立有效的沟通渠道。

2. E2E集成测试设计

E2E测试设计包括在测试环境下定义集成系统的任务和定义测试程序的任务。可以从两个方面来定义被测的集成系统：E2E功能视图；结构视图，包括物理结构和逻辑结构。下面通过一个细线程树及所附条件来介绍被测系统规范。

细线程(Thin Thread)的定义：一个完整的数据或消息的踪迹，使用最低限度的具有代表性的外部输入数据样本，通过系统内部的相互连接部分的转换，产生最低限度的具有代表性的外部输出数据样本。细线程主要是用来阐释某一方法具有指定的功能[5]。

细线程是集成系统中最小的场景，从最终用户的角度看，它是一个完整的场景，系统接收输入数据，经过计算，并输出处理的结果。它描述了整个场景，而且只描述一个功能。

一些共享某些公共数据的细线程可以组成一个细线程组。这些组具有层次结构。细线程组中的所有细线程和子细线程组可以构成一棵细线程树。树的根是整个被测的集成系统，它的分支节点代表相关的细线程集合，它的叶子代表一个具体的细线程。在同一组中的细线程通过它们共同的功能相关联。

包含关系的定义：一条细线程A的执行路径可以是另一条细线程B的一部分，称细线程A是B的子细线程，B依赖于A。相同关系的定义：一条细线程A具有和另一个细线程B相同的执行路径。在这种情况下，A和B共享某些属性，如条件等；这些细线程之间的关系可以用来安排测试用例的执行。如果某条细线程处在系统关键路径上，那么它应该尽早地、完整地、恰当地被测试。如果要选定一些细线程进行测试，则应该选择那些彼此执行路径相互独立的细线程，从而保证一定程度的覆盖。

细线程树构造是一个交互式进程包括如下一些活动：确定和指定细线程、确定和指定与细线程相关的条件、将细线程和条件组织到树中以及完成完整性和一致性校检。基于细线程树和条件树规范生成测试用例，分析每个细线程的风险，分析每个细线程的用途，为了执行制定测试用例时间进度。

3. E2E集成测试规格书

E2E测试规格书是E2E测试的核心，用来表述系统需求。从最终用户的角度出发，用情景描述系统行为：包括正常的输入，非正常输入，常规案例和特殊处理。E2E测试规格书是一种半规范化，有层次的构架，含有用例生成的相关数据，并可以追溯到其他软件工件上如系统需求及设计。E2E测试规格书有主要有两部分信息：细线程和条件。

细线程的定义模板所包含的内容有：ID、名字、描述、输入/输出、前提条件/后续条件、隐藏的成分、状态、代理以及风险。一组具有特定公共部分的细线程的集合形成了细线程组

(thin thread group)；细线程组递归分组，重组为一个树结构。细线程树可以自顶向下构建，按功能性进行分解；也可以自低向上构建，利用提取和合成。

细线程之间的关系如下：

(1) 细线程具有独立的执行路径。

(2) 细线程具有隐藏的执行路径。

(3) 细线程具有相同的执行路径。

条件是断言，要激活这个功能条件必须是真。条件的例子有数据条件、信息条件、环境条件以及系统状态。条件分析是完全性分析和一致性分析的一部分，可以发现新的细线程，通过发现不完全/不一致的条件，发现附属于矛盾条件的细线程。

条件定义模板包括的内容有：ID、名字、描述、受影响的细线程等。条件组是一组具有特定公共属性的条件的集合。递归分组，可组为一个树结构。条件树可以自顶向下分解，自底向上提取和合成的方式构建。条件间的关系有：独立条件、互斥条件、引发/被引发的条件。相关条件。

从细线程或复杂场景生成测试用例。确定相关的子系统，包括软件和硬件系统，确定为线程输入的数据。基于不同的测试技术，利用满足结合细线程的条件的输入数据，根据对于细线程的描述来决定预期的结果。

4. E2E 集成测试风险分析

风险分析是系统和软件开发中的一项重要活动。基于风险分析，系统的临界条件可以得到彻底的测试。当资源有限的时候，测试应该把那些重要的细线程放在首位。一种排列细线程的方法是至少基于两种因素($Probability_{thin\text{-}thread}$，$Conseqaence_{thin\text{-}thread}$)给每一个细线程分配一个风险。

下面几个构件常有很高的失败可能性：

- 在先前的模块或综合测试中就显现出不可靠性的成分。
- 具有复杂的实现过程或者具有合并的复杂功能的成分。
- 与许多其他成分相连的成分。
- 由于错误或功能的变更而最近刚刚修改过的成分。

如果一个细线程可能危害到系统的整体任务的完成或者对环境造成重大损害，则它必须被多次测试。一个细线程的风险其实是一个关于它的失败可能性和它的失败结果的方程：

$$Risk_{thin\text{-}thread}=F(probability_{thin\text{-}thread},Consequence_{thin\text{-}thread})$$

一个条件的风险可以被估计为：

$$Risk_{condition}=F(probability_{condition},Consequence_{condition})$$

E2E 的测试用例是基于一个细线程以及它的条件生成的。测试用例风险可以基于这个细线程的风险和它的条件风险来估计：

$$Risk_{test\text{-}case}=F(probability_{test\text{-}case},Consequence_{test\text{-}case})$$

一个细线程的风险分配是动态的。根据程序的执行，细线程的值是变化的。

在一个测试项目的开始，失败的可能性是很大的，几乎所有的细线程都有很高的风险。当系统得到了测试和纠正，它的失败的可能性可能会降低，并且，一些细线程最终可以具有最小的风险。对于那些失败可能性很高的细线程称为高因果关系细线程；如此而言，即使它们的失败可能性降低，它们的风险仍然很高。在软件修改期间常常会引入缺陷；因此，对于那些执行在变化的成分的细线程的风险分配而言，在软件修改之后风险将会提高。

1）怎样选择输入

测试输入是无限的，所以要挑选测试输入。挑选条件边界附近的点进行测试（边界测试）。随机挑选输入（随机测试）。把输入分成等价的几类，然后在这几类中挑选典型数字（等价类划分）。根据用处产生测试用例（基于用途测试）

2）怎样选择输出

与选择输入相比，输出要相对复杂得多。要根据输入确定输出可能要经历很多的步骤和过程，这其中包括计算和非计算，并且确定了输入也未必就能完全确定输出。例如：对于傅里叶级数

$$x(t)=a_0+a_1\cos(w_ot+q_1)+a_2\cos(2w_ot+q_2)+\cdots+a_N\cos(Nw_ot+q_N)$$

即使确定了输入，但 $x(t)$ 仍然是随 t 的改变而改变，因此很难确定其输出。当然，可以采取一定的方法达到测试的目的，例如，选取特定的 t 时刻，使得对于 $i\in[1,N]$，iw_ot+q_i 为特殊角。

在图形用户界面中，期待的输出包括屏幕快照以及视窗位置和标题。

5. E2E 测试执行

E2E 测试执行准备。在测试之前，测试工程师要识别出以下组件：要测的子系统；支持系统，包括硬件、固件、数据库、第 3 方组件，确保子系统合适地执行；备份系统和程序，以防万一测试损坏了系统，系统还可以恢复；测试数据，包括测试输入数据、数据库、执行测试用例所需要的文件；测试工具，包括自动输入数据生成工具、测试驱动、测试结果记录工具；测试组。

在模拟环境下进行 E2E 测试包括以下步骤：开发或取得模拟程序，把模拟程序的参数调成跟要测试的系统一样；执行应用系统；选择测试用例，生成输入数据，记录执行结果；重复选择和执行测试用例的过程，在必要的时候恢复系统和模拟状态，直到所有计划的测试用例都被执行了为止。

在操作环境中进行 E2E 测试包括以下步骤：建立环境，调用应用系统，选择测试用例，根据系统外部接口生成输入数据；重复选择和执行测试用例的过程，在必要的时候恢复系统和模拟状态，直到所有安排的测试用例都被执行了为止。

测试结果标准是所有的测试用例都被执行，测试覆盖率要求得到满足，一定已经获得确认。在 E2E 测试执行的时候需要加入文档的结果有：选择的测试用例和需要测试用例测试的需求项，测试用例的输入数据，测试用例的输出；每个子系统的接口状态，包括子系统

之间通过接口交换的数据和子系统和支持系统之间交换的数据；子系统的状态，包括硬件、软件、支持系统。

6. 测试结果分析

1）缺陷识别和改正

缺陷是当执行的时候可能会产生不正确结果的代码错误。要找出缺陷，测试输出要跟预期输出进行比较。在测试过程中缺陷识别可以被优先并改正[6]。

2）评估涟漪效应

软件工件(Work Produet)的一部分改动会影响其他相关部分的现象叫做波漪效应(Ripple Effect Analysis,REA)，而迭代的分析并去除改变的副作用的过程就叫做涟漪效应分析。波漪效应分析过程包括以下步骤：

(1) 提出软件修改。

(2) 识别依赖于被改变部分的其他模块。

(3) 决定是否要改变依赖部分来保持一致性。

(4) 如果需要改变，则从第(1)步开始循环进行 REA。

(5) 如果不需要改变，停止并等待修改软件。

REA 的过程就像生成树，它的终端结点是不需要改变但依赖于前面部分的软件模块。在细节上，REA 过程不会具体到特殊程序语言或设计范式。比较特殊的是，REA 可以用来维持细线程树和条件树的一致性。

3）评估测试覆盖率。

如果出现以下情况，就需要额外的测试：

(1) 系统的一个片断有很多错误，这常意味着这个片断倾向于有错误并需要额外测试。

(2) 当缺陷被修改了后，被修改的系统要求额外测试来保证错误已经被消除而且没有新的错误引进。

(3) 当一个新的功能特性被改变后，需要额外测试来保证没有新的错误引进。

(4) 当软件不符合结束标准，需要额外测试。

4）回归测试

回归测试是在修改过的软件上重新运行测试用例，普遍用于软件程序被修改的情况。一个要点就是回归测试只能保证这些应该保留的部分保持不变。需要在不同的层面上进行回归测试，首先是模块测试层面，之后是在集成测试层面，最后是终端到终端测试层面。在回归分析的每一个层面，测试人员有不同的任务。

(1) 测试用例确定。

(2) 测试用例再确认。

(3) 测试用例执行。

(4) 通过检验测试结果来确认失误。

(5) 确认差错并改正。

若要开发新的测试用例,首先要根据修改后的系统和需求重新评估细线程树和条件树,根据需要重新组织,扩大或者剪切细线程树;之后,根据修改后的细线程数和条件数重新评估测试用例。在开发新测试用例时,首先是针对错误区域或错误功能点,其次如果需求发生变化,则针对修改过的功能点或添加的新功能。

测试都需要效率评估,E2E 测试也不例外。测试的目的是发现错误和缺陷,因此,发现错误和缺陷的数目越多,测试的效率越高。评估测试效率的标准主要有:

(1) 发现缺陷的数目/测试用例的使用数目。

(2) 通过的细线程数量/测试的细线程数量。

(3) 发现错误的数量/测试用例的使用数量。

7. 依赖关联分析

依赖关联分析为回归测试和涟漪效应分析提供了基础。涟漪效应分析(REA)使用可信赖的消息确认需要哪些额外的变化来保证所有的组件保持一致。

- 功能可信度。
- 输入可信度。
- 输出可信度。
- 输入/输出可信度。
- 持久数据可信度。
- 执行可信度。
- 条件可信度。

8. 远程测试(Remote Testing)

由于分布式系统的异步通信,加上系统的组成分布在不同的地方,分布式系统和设备的测试非常困难,测试人员可以用测试代理通过网络协议进行通信。

图 5-4 为分布式测试示意图,步骤描述如下:

(1) 注册。测试中心注册流程。

(2) 获取测试用例。测试经理启动一个测试流程/测试计划,其中包括从数据库中获取一组测试用例。

(3) 测试调度。测试经理将当前测试计划的测试用例以及可用的测试中心的信息发送到测试调度。

(4) 将测试用例传送到测试中心。测试调度将测试用例发至已协调好的测试中心。

(5) 传送测试用例。测试用心将测试用例传送至测试代理。

(6) 访问 SUT(待测系统)。测试代理将测试用例转化为测试脚本,并将这些脚本发至 SUT 以供执行。

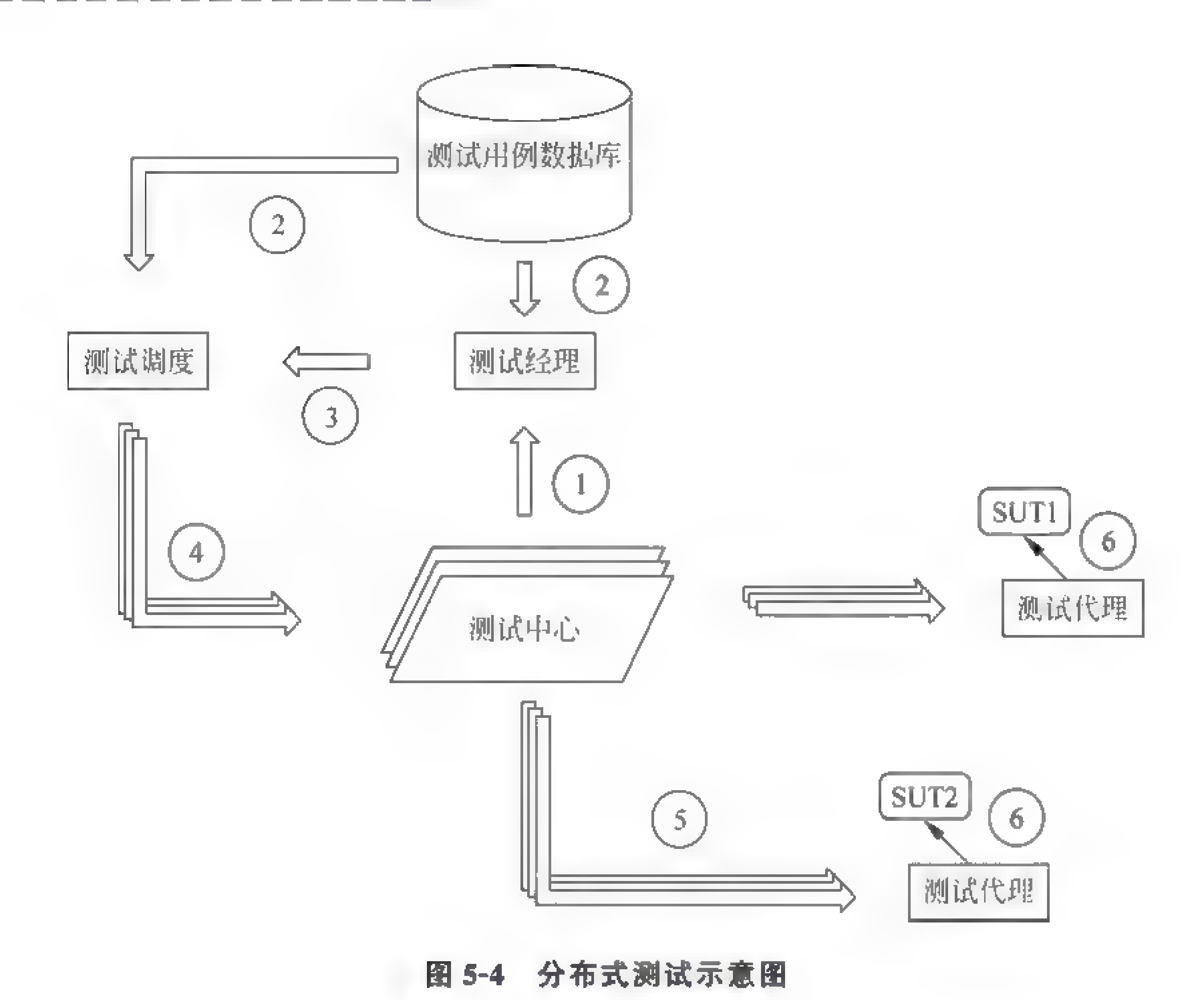


图 5-4 分布式测试示意图

5.3 总结

单元测试是测试的基础级别。单元测试着眼于程序或系统的较小组件模块，执行每个模块以证实其履行了指定功能的过程。单元测试的优势在于它容许对小单元的测试和调试，因此为管理从小单元到大单元的集成过程提供了更好的方式。把组件聚合后，必须通过测试确认所有的组件之间正确地协作运行。因此集成测试的目标是暴露接口的缺陷，以及聚合后的组件之间相互作用的缺陷。E2E 集成测试是从终端用户的角度对系统或应用程序进行端到端的功能测试，目的是验证相互连接的系统形成的集成系统是否能完成终端用户的工作目标及其任务。

5.4 参考文献

[1] R. Pressman. *Software Engineering: A Practitioner's Approach*. Boston: McGraw Hill, 2005
[2] E. Yourdon. *Techniques of Program Structure and Design*. Prentice-Hall, 1975
[3] IEEE standard for software unit testing, pp. 1008～1987

[4] Raymond Paul. *End-to-End Integration Testing*. *Second Asia Pacific Conference on Quality Software* (*APAQS'01*). p. 211

[5] W. T. Tsai, Xiaoying Bai, Ray Paul. End-To-End Integration Testing Design. *25th Annual International Computer Software and Applications Conference* (*COMPSAC'01*). pp. 166～171

[6] W. T. Tsai, Xiaoying Bai, Ray J. Paul, Lian Yu. Scenario-Based Functional Regression Testing. *COMPSAC 2001*: pp. 496-501

5.5 思考与练习

1. 单元测试的特点是什么？"单元"指的是什么？单元测试对构件的5方面进行测试，这5个方面指的是什么？
2. 单元测试的规程是什么？单元测试有什么局限性？
3. 集成测试与单元测试的区别是什么？
4. 集成测试策略有哪两种？各有什么特点？
5. 什么是端到端的集成测试？简述其过程。

5.6 进一步阅读

C. Kaner, J. Bach, B. Pettichord. Lessons. Learned in Software Testing. John Wiley & Sons, 2002

L. Copeland. A Practitioner's Guide to Software Test Design. Boston: Artech House Publishers, 2004

R. D. Craig, S. P. Jaskiel. Systematic Software Testing. Norwood, MA: Artech House Publishers, 2002

Control your test-environment with DbUnit and Anthill http://www-128.ibm.com/developerworks/library/j-dbunit.html

Adrew Hunt 著，陈伟柱译. 单元测试之道 Java 版. 北京：电子工业出版社

第6章

JUnit测试工具

JUnit是一个开源的Java编程语言的单元测试框架，最初由Erich Gamma和Kent Beck编写。JUnit在代码驱动的单元测试框架家族里无疑是最为成功的一例。多次获得JavaWorld Editors' Choice Awards中的Best Java Performance Monitoring/Testing Tool奖。

使用JUnit框架需要继承TestCase类(JUnit 4以前)，用Java语言来编写自动执行、自动验证的测试。这些测试在JUnit中称作"测试用例(Test Case)"。JUnit能够把相关测试用例组合到一起，称之为"测试套件(Test Suite)"。JUnit还提供了一个"运行器"来执行一个测试套件。如果有测试失败了，这个测试运行器就报告出来；如果没有失败，就会显示OK。

具有JUnit经验对于应用"测试驱动开发(TDD)"的程序开发模型是非常重要的。在讨论测试驱动开发时，常常会涉及JUnit的一些知识。测试驱动开发是一种编程风格，这种风格的大致含义是：先编写测试代码，再编写产品代码本身，另外还需要在编写代码的时候执行重构。这样的产品代码测试覆盖率高，容易修改，容易扩展，并且容易理解。本章对"测试驱动开发"不做深入讨论，感兴趣的读者可以参考文献[1]。

下面简要概述使用JUnit的益处。

(1) 提高开发速度：测试是以自动化方式执行的，提升了测试代码的执行效率。

(2) 提高软件代码质量：它使用小版本发布，控制代码更改量，便于开发人员排除错误。同时引入重构概念，让代码更干净和富有弹性。

(3) 提升系统的可信赖度：作为回归测试的一种实施方式。支持修复或更正后的"再测试"，可确保代码的正确性。

(4) JUnit和Ant的结合可以实施增量开发和自动化测试。

(5) 与IDE的集成：由于JUnit拥有广泛的影响力，主流Java IDE都对其提供了良好的支持。

本章将介绍JUnit的使用、框架设计和相关的"模仿对象(Mock Objects)"以及数据库的单元测试工具DbUnit。介绍JUnit是用JUnit 3.8版本，在本章结束之前还将简要介绍JUnit 4。

6.1 使用JUnit

用JUnit编写一个测试用例；安装JUnit，运行JUnit测试。

6.1.1 一个简单的例子

用JUnit编写一个简单的测试，可以通过以下3个简单步骤：

(1) 创建TestCase类的一个子类。

(2) 编写若干测试用例，每个测试用例写成如下格式的子类方法，注意JUnit对于测试用例的命名法是test＋＜TestCaseName＞测试用例的名字。

```
public void test<TestCaseName>(){…}
```

(3) 编写一个测试套件方法用来把第(2)步中编写的测试用例加入到测试套件中。

```
public static Test suite(){…}
```

然后编译上述子类以及被测构件，用JUnit提供的运行器TestRunner运行测试。

程序清单6-1是一个简单的JUnit的测试例子。这个简单测试的目的是：

(1) 验证Java标准类库的Math类中的max()方法。

(2) 测试被零除的结果。

下面来看一下列表中的要点：

- 程序开始时是通过继承(在Java里用extends保留字)TestCase类，来创建的一个子类SimpleTest。
- SimpleTest中有两个测试：testMax()(第2～5行)和testDivideByZero()(第6～9行)。这两个测试的命名规则是遵循JUnit的惯例，即测试方法的名字是以test开头的。测试方法的可见性都必须是公共的(public)，而且必须无返回值，即返回类型为viod(见第2行和第6行)。在testMax()中，在第3行局部变量x利用Math.max(5,10)得到5和10的最大值，表达式x>=5&&x>=10的预期结果是true。利用JUnit框架提供的断言(Assertion)方法assertTrue(boolean)来帮助验证Math.max(5,10)的结果(见第4行)。运行TestRunner会给出结论。在testDivideByZero()中，在第8行整数被零除，下面来看一下在后面的图6-4中将给出JUnit框架执行的结果。

- 编写测试套件方法 suite()，它是公共的、静态的(static)，返回类型是 Test(对于 Test 类型，后面还会详细讨论)。利用 TestSuite 的 TestSuite(SimpleTest.class)建造，把 SimpleTest 的两个测试方法 testMax()和 testDivideByZero() 放入 suite()里。把各个测试放入 suite()还有另外一种方法，后面将会详细介绍。

清单 6-1：一个简单的 JUnit 示例

```
public class SimpleTest extends TestCase {
    public void testMax() {
        int x=Math.max(5,10);
        assertTrue(x>=5 && x>=10);
    }
    public void testDivideByZero() {
        int zero=0;
        int result=8/zero;
    }
    public static Test suite() {
        return new TestSuite (SimpleTest.class);
    }
}
```

下面一节将介绍 JUnit 的安装，并以清单 6-1 为例说明如何运行 JUnit 测试程序。

6.1.2 JUnit 安装与运行

要安装和使用 JUnit 是很容易的，只需 3 个步骤：

(1) 下载 JUnit 软件。

(2) 将 JUnit 包解开，放到文件系统中。

(3) 运行 JUnit 测试时，将 JUnit 中的所有 *.jar 文件放到类路径中。

1. 下载 JUnit

目前，下载 JUnit 的最好地方是 http://www.junit.org。在此，会有一个下载链接，链到此产品的最新版本。单击下载链接，将这个软件下载到用户机器的文件系统中。

2. 解包 JUnit

运用可获得的解包工具，将 JUnit 包解开，放到文件系统一个目录下。表 6-1 列出了 JUnit 发布中的一些主要文件和目录。

表 6-1 JUnit 发布里所含文件与目录

文件/目录	描 述
junit. jar	JUnit 框架结构、扩展和测试运行器的二进制发布
src. jar	JUnit 的源代码，包括一个 Ant 的 buildfile 文件
junit	是个目录，内有 JUnit 自带的用 JUnit 编写的测试示例程序
javadoc	JUnit 完整的 API 文档
doc	一些文档和文章，包括 Test Infected: Programmers Love Writing Tests 和其他一些资料，可以帮助用户入门

3. 检验安装 JUnit

要检验安装是否正确，可以通过执行 JUnit 发布中自带的、用 JUnit 编写的测试示例程序。要执行这些程序，应按照以下步骤进行：

(1) 打开一个有命令行提示的窗口。

(2) 转到含有 JUnit 的目录下(Windows 系统的 C:\junit3. 8. 1，或 Linux 系统的/opt/junit3. 8. 1 或任何安装时选定的地方)。

(3) 执行下列命令：

```
>java -classpath junit. jar; . Junit. textui. TestRunner junit. samples. AllTests
```

然后，会看到类似于图 6-1 的结果。

```
C:\junit3.8.1>java -cp junit.jar;. junit.textui.TestRunner junit.samples.AllTest
s
.......................................
.......................................
.......................................
Time: 1.39

OK (119 tests)

C:\junit3.8.1>
```

图 6-1 Windows 平台下 JUnit 的安装结果

对于每个测试，测试运行器都会打印出一个点，让我们知道现在正在执行的进度。在执行完所有的测试之后，测试运行器会显示 OK，并且告诉我们它总共执行了多少测试和花费了多少时间。

在上述执行测试的命令中，类路径包含了 junit. jar 和当前的目录(.)。junit. jar 是仅有的一个需要放到类路径下的文件。当前的目录(.)正是解包 JUnit 的那个目录，JUnit 测试的

所有*.class文件所在目录从此目录开始。接着的一个参数——junit.textui.TestRunner，是JUnit的基于文本的测试运行器的类名。这个运行器会执行所有的JUnit测试，并将结果报告给控制台。最后一个参数——junit.tests.AllTests，是需要运行的测试套件的名字。

设置*.jar路径运行JUnit测试时，将JUnit中的所有*.jar文件放到类路径中。有两种方法来设置JUnit的类路径：一是将JUnit中的所有*.jar文件放到类路径中；二是运行JUnit测试程序时，在运行命令选择项里指定如图6-1所示。

4. 运行JUnit测试

JUnit框架提供两种运行测试的方式：文本式和图形用户界面式。在介绍JUnit安装检验时，使用的是文本式，即执行命令是从命令行输入的，结果显示到控制台。下面以清单6-1为例说明如何运行图形用户界面式的JUnit测试运行器。

在运行JUnit的测试程序之前，需要编译。编译的命令如下：

```
>javac -classpath junit.jar; <your_classpath> -d <destination_directory> <your_source_ files>
```

和运行JUnit的测试程序一样，编译时需要将junit.jar这个文件放到类路径下。然后指定我们自己的类路径，可选项-d容许把源程序编译成.class文件后放到指定的目录里。

JUnit框架提供图形用户界面式运行测试有两个版本：AWT(Abstract Window Toolkit)版本和Swing版本。使用AWT版本时输入以下命令来启动JUnit运行器：

```
>java -cp junit.jar; <your_classpath> junit.awtui.TestRunner
```

命令启动后出现如图6-2所示的AWT版本图形用户界面。对于界面的5个部分简单介绍如下：

(1) 测试类名。这个名是TestCase测试类全称(Fully Qualified)，如对于AllTests来说，是junit.samples.AllTests。

(2) 结果显示横条。如果测试无错误而且无失败，则横条显示"绿"色；否则横条显示"红"色。横条上面的选项容许选择每次运行测试时重新装入测试的字节码类文件。如果要动态修改测试程序，那么这个选项很有用，它让我们能运行最新的测试版本。

(3) 结果统计。界面显示3个测试统计结果，即测试套件里总共测试方法的数目，测试发生错误的数目和测试失败的数目。关于"测试错误"与"测试失败"的定义与理解，在后面还会继续讨论。

(4) 错误与失败的显示。这部分有两个显示区，上面的显示区是错误和失败的列表(如果有的话)；下面的显示区是从错误和失败的列表中选定项的详细说明。

(5) 信息显示栏：显示运行测试套件的所有测试方法的时间，以秒为单位。



图 6-2 AWT 版本的 JUnit 运行器界面

使用 Swing 版本时输入以下命令来启动 JUnit 运行器：

>java -cp junit.jar; <your_classpath> junit.swingui.TestRunner

命令启动后出现如图 6-3 所示的 Swing 版本图形用户界面。Swing 版本界面除了外观比 AWT 版本表达丰富外，Swing 界面还容许浏览文件目录，从而方便选择测试套件(如图 6-3 右方图所示)。

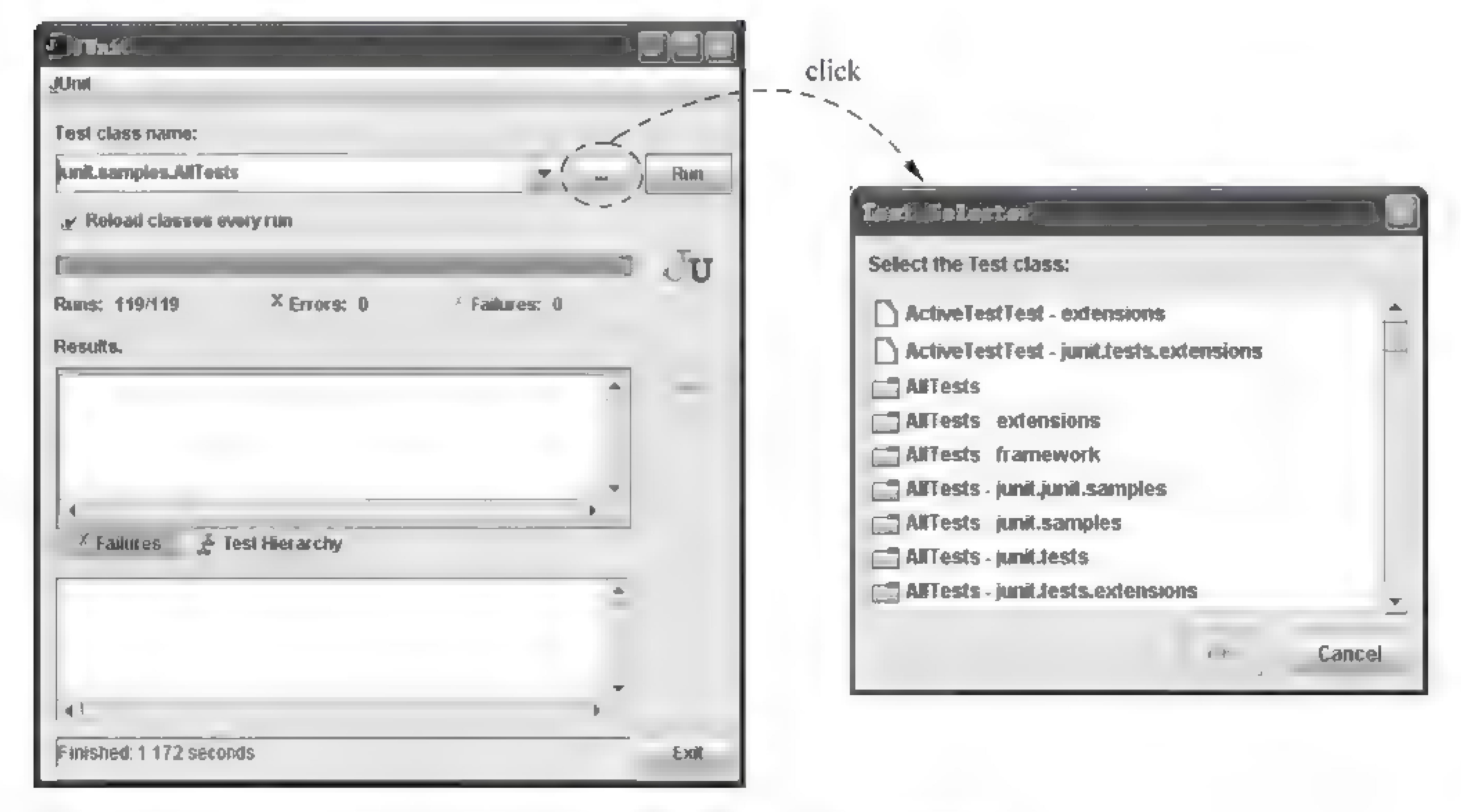


图 6-3 Swing 版本的 JUnit 运行器界面

现在编译并运行清单 6-1 所示的测试程序。假设把 SimpleTest 程序放到当前目录下的 junit 目录下的 samples 目录里。如图 6-4 所示，用 AW 版本的 JUnit 运行器界面来显示程序 SimpleTest 运行的结果。测试统计报告，运行两个测试方法，有一个错误，无失败。同时横条显示“红”色。错误与失败的两个显示区分别列出有错误的测试方法 testDividedByZero()和显示错误的详细信息。底部的信息栏里显示选中的出错的测试方法。



图 6-4 SimpleTest 程序运行结果

界面上还有 3 个按钮没有解释。右边第 1 个按钮 Run：当输入测试类名后，按此按钮来执行测试。右边第 2 个按钮 Run：如果发生错误或失败时，可以对发生错误或失败的测试方法进行修改、编译，然后按此按钮进行再测试。注意，测试类名下方的选择项必须被选中。Exit 为退出按钮。

6.1.3 JUnit 常见问题

1. JUnit 的断言

先来看一下：在清单 6-1 SimpleTest 中再加两个测试方法 testAdd()和 testEqual()，如清单 6-2 所示。

清单 6-2：用断言方法“测试”失败

```
public void testAdd() {
        double result=2+3;
```

```
        assertTrue(result==6);
}
public void testEqual() {
        double a=6,b=7;
        assertTrue( a==b);
}
```

重新编译 SimpleTest 并运行，如图 6-5 所示。

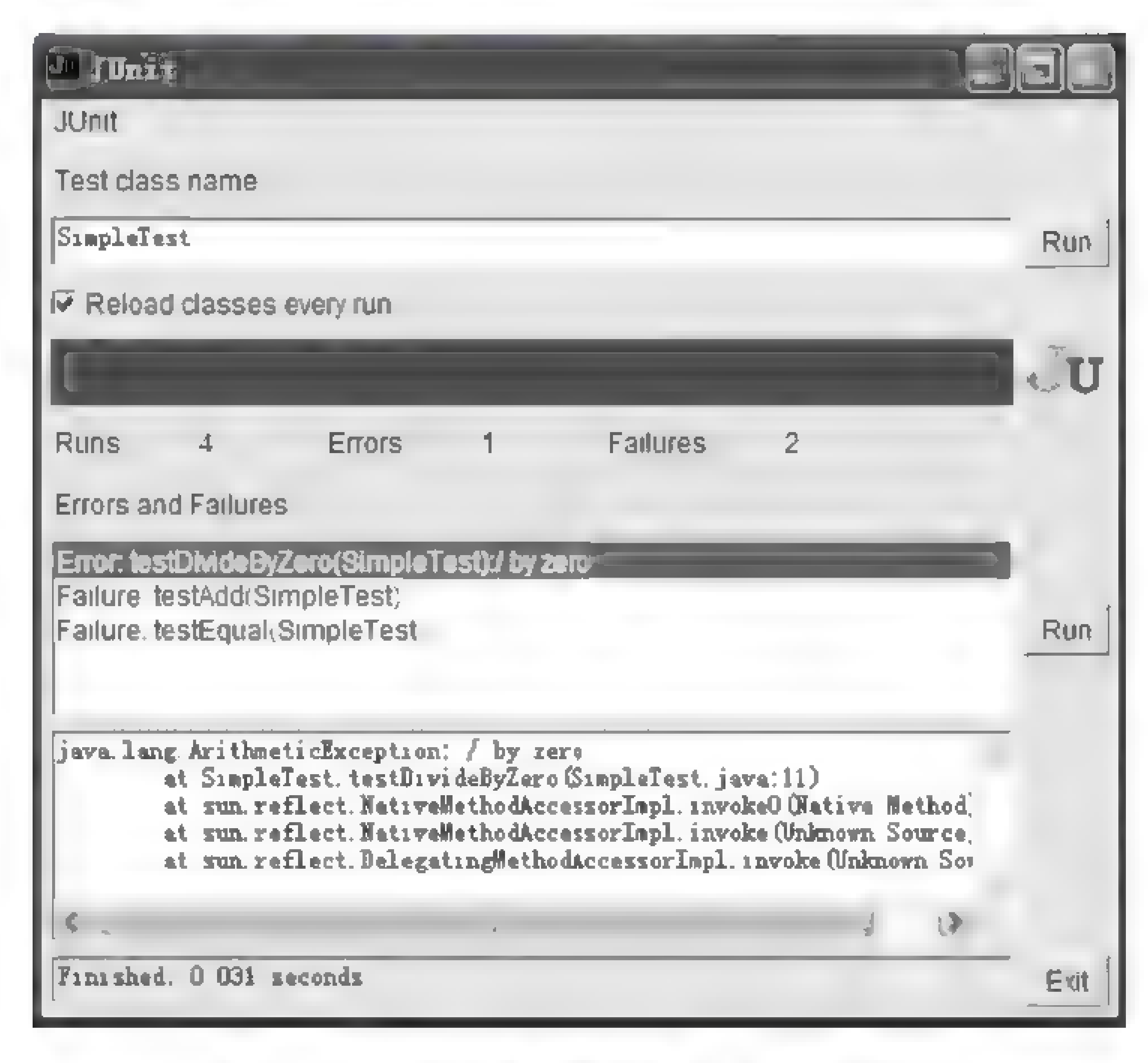


图 6-5 SimpleTest 程序运行结果：有错误与失败

要理解 JUnit 是怎样决定一个测试到底是通过了还是失败了，就需要了解断言的方法是怎样表示一个断言已经失败了的。要让 JUnit 测试能够自行验证，就必须对于对象的状态做出断言。当 JUnit 断言失败了，断言的方法就会抛出一个异常，来表示这个断言失败了。更准确地说，当断言的方法失败时，断言的方法会抛出一个错误：AssertionFailedError。下面是 assertTrue()的源代码。

清单 6-3：assertTrue()的源代码

```
static public void assertTrue(boolean condition) {
        if (! condition)
                throw new AssertionFailedError();
}
```

当 assertTrue()传入的条件是 false 时,这个方法会抛出 AssertionFailedError,来表示这是个失败的断言。JUnit 框架捕获这个错误,将其标记为"失败",并记住是哪个测试失败了,然后再进行下一个测试。当 assertTrue()传入的条件是 true 时,JUnit 认为该测试通过了。

当一个断言失败了,加上一个简短的消息来通知这个失败的一些属性甚至是失败的原因,可能会产生比较好的效果。在 JUnit 框架中,一个断言的方法的第一个参数作为可选项,接受一个 String 类型的参数,包含一些消息,在断言失败的时候显示出来。assertTrue()有两种形式:

```
static public void assertTrue(boolean condition)
static public void assertTrue(String message,boolean condition)
```

在 JUnit 框架中,和 assertTrue()相对应的断言的方法有 assertFalse(),它也有两种形式:

```
static public void assertFalse(boolean condition)
static public void assertFalse(String message,boolean condition)
```

和 assertTrue()相反,当 assertFalse ()传入的条件是 false 时,JUnit 认为该测试通过了;当 assertFalse ()传入的条件是 true 时,会抛出 AssertionFailedError,来表示这个断言失败了。

JUnit 框架提供一套断言的方法,以用于不同种情况。感兴趣或需要使用这些断言的方法的读者,可查阅 JUnit 文档或参考文献[2]和[3]。

2. 失败与错误

JUnit 的测试结果中"失败"与"错误"是不同的。"失败"是指断言失败,而"错误"是指当某种其他异常发生时,而这种异常还没被测出也没被预料到。这种区别是微妙的也是有用的。出现"失败"的断言通常表示产品代码中有问题,而出现"错误"却表示测试本身或周围的环境存在着问题。

"错误"可能是得到一个不正确的异常,或者调用一个空引用上的方法(抛出 NullPointerException),或者对于数组的操作超出数组的范围(抛出 ArrayIndexOutOfBoundsException);也可能是磁盘已经满了或者网络连接不通,或者一个文件找不到。JUnit 并不把这些算作产品代码的缺陷,而是采取一种"举手投降"方式来表达:"有些不对劲了。我不能分辨这个测试是否通过。请解决这个问题后再重新测试一次。"

有时候,JUnit 框架的"失败"被称为"预期的失败条件";而"错误" 被称为"不曾预料到的失败条件"。测试人员用断言方法"测试"失败。如清单 6-2 所示的两个测试,testAdd()和 testEqual()分别用到了断言方法 assertTrue(boolean),我们期望返回的值是 false,即两个断言方法"失败"了,JUnit 的运行器将报告这两个"失败"。

然而"错误"，即"不曾预料到的失败条件"，在运行时候(runtime)将引起异常。如清单 6 1 所示，第 8 行语句将引起异常，JUnit 的运行器将报告这为"错误"。所谓"不曾预料到的"是指无法用 JUnit 提供的断言方法来预期其结果，或者是测试人员事先没料到的测试程序不当之处。有人可能要问："清单 6 1 的第 8 行不是有错吗？不是能'预期'到这个错误吗?"清单 6 1 是 一个示例，为了便于解说，故意把那个错写得很明显。在实际写测试程序时，犯错误的地方往往很隐蔽，很难发现。

当得到 一个测试运行结果的报告时，如果报告里既有"失败"又有"错误"时，建议先调查"错误"，解决了错误问题以后再重新运行这个测试。

3. 测试套件

在清单 6-1 中，利用 TestSuite 的构造函数 TestSuite (SimpleTest. class)，把测试程序 SimpleTest 的两个测试方法 testMax()和 testDivideByZero()都加入到测试套件里。清单 6-1 中的 suite() 方法代码：

```
public static Test suite() {
        return new TestSuite (SimpleTest.class);
}
```

和下面的代码是等价的，也就是说，上面的代码可以用下列代码代替：

```
public static Test suite() {
        TestSuite suite=new TestSuite();
        suite.addTest(new TestSuite ("testMax"));
        suite.addTest(new TestSuite ("testDividedByZero"));
        return suite;
}
```

如果在 SimpleTest 中增加另外的测试，例如 testAdd()和 testEqual()，那么 TestSuite (SimpleTest. class)会自动将这两个测试加到测试套件里，而无须手动将它们写入测试套件。

其实，如果在清单 6 1 中不定义 一个 TestSuite，JUnit 测试运行器将自动生成 一个默认 TestSuite。这个默认 TestSuite 扫描测试类中以 test 开头的所有方法，在内部为每个 testXXX 方法生成 一个 TestCase 实例，并把被调用的方法名作为参数转入 TestCase 的构造函数。

现在的情况是： 一个测试类里有很多测试方法，我们并不想运行所有这些方法，而只是其中 一些。这样就不能再用清单 6 1 中的方式 TestSuite(SimpleTest. class)来写测试套件，也不能用默认 TestSuite。这种情况下，可以使用"手工(Manual)"方式。比如我们只是想运行测试清单 6 1 中 testDividedByZero()，则可以用以下代码代替清单 6 1 的相应部分，如清单 6 4 所示。把 testDividedByZero()加入 TestSuite 是通过使用 Java 的匿名内部类(Anonymous Inner

Classes)方式实现的(见清单 6-4 的第 3 和第 4 行)。

如果用匿名内部类方式把测试方法加入测试套件，那么这些测试方法的命名规则不用遵循 testXXX 方式。但是在其他场合下，如果希望运行测试类里的所有测试方法，那么还是遵循 JUnit 的命名规则比较方便。

清单 6-4：手工方式加入测试套件

```
1.  public static Test suite() {
2.          TestSuite suite=new TestSuite();
3.          suite.addTest(new SimpleTest ("testDivideByZero")
4.                  {protected void runTest(){testDivideByZero(); }});
5.          return suite;
6.  }
```

4. 测试置具

几个测试可能都用到同一个或同一组对象，如果在每个测试里各自编写这样的对象，将造成代码冗余，不利于维护。应该把这样的代码分离出来单独写，让所有的测试都可以利用这些对象代码。在 JUnit 框架中，把这些对象称为"测试置具(Test Fixture)"。清单 6-5 给出了一个例子，说明如何使用测试置具。

(1) 首先将 fEmpty 和 fFull 声明为实例变量(Instance Variable)，以便它们可以被 VectorTest 类的方法引用。

(2) 在 setUp()方法里，fEmpty 和 fFull 被实例化为 Vector 类的两个对象(严格地说，是对象引用)。fEmpty 是一个空 Vector，而 fFull 里加入 3 个整数，分别为 1、2 和 3。

(3) 测试方法 testCapacity()利用 fFull 作为测试置具，先对其求 Vector 的大小放入变量 size 中，然后再加入 100 个整数，最后用 assertTrue(fFull.size()==100+size)断言。

(4) 测试方法 testContains()又一次利用测试置具 fFull，通过使用 assertTrue(fFull.contains(new Integer(1)))断言，来测试 fFull 是含有整数 1；利用测试置具 fEmpty，通过使用 assertTrue(! fEmpty.contains(new Integer(1)))断言，来测试 fFull 是不含有整数 1。

读者先想一想，清单 6-5 的运行结果是什么，然后编译并用 TestRunner 运行一下，以验证自己的想法。

清单 6-5：利用测试置具示例

```
public class VectorTest extends TestCase {
        protected Vector fEmpty,fFull; //测试置具 ①

        //设置测试置具
        protected void setUp() {        ②
```

```
        fEmpty=new Vector();
        fFull=new Vector();
        fFull.addElement(new Integer(1));
        fFull.addElement(new Integer(2));
        fFull.addElement(new Integer(3));
    }
    public void testCapacity() {       ③
        int size=fFull.size();
        for (int i=0; i<100; i++)
                fFull.addElement(new Integer(i));
        assertTrue(fFull.size()==100+size);
    }
    public void testContains() {       ④
        assertTrue(fFull.contains(new Integer(1)));
        assertTrue(! fEmpty.contains(new Integer(1)));
    }
}
```

6.1.4 一个自动售货机的例子

自动售货机(Vending Machine)可以销售多种产品，如冷饮料、热咖啡、糖果、小吃，甚至速冻食品等，自动售货机可以方便地安装在城市的街道、学校、工厂中，给人们的生活带来方便。

自动售货机用途很广，从硬件构成上看，主要有投币口、退币口、物架及产品、标有价格的按钮和取品处，如图 6-6 所示。

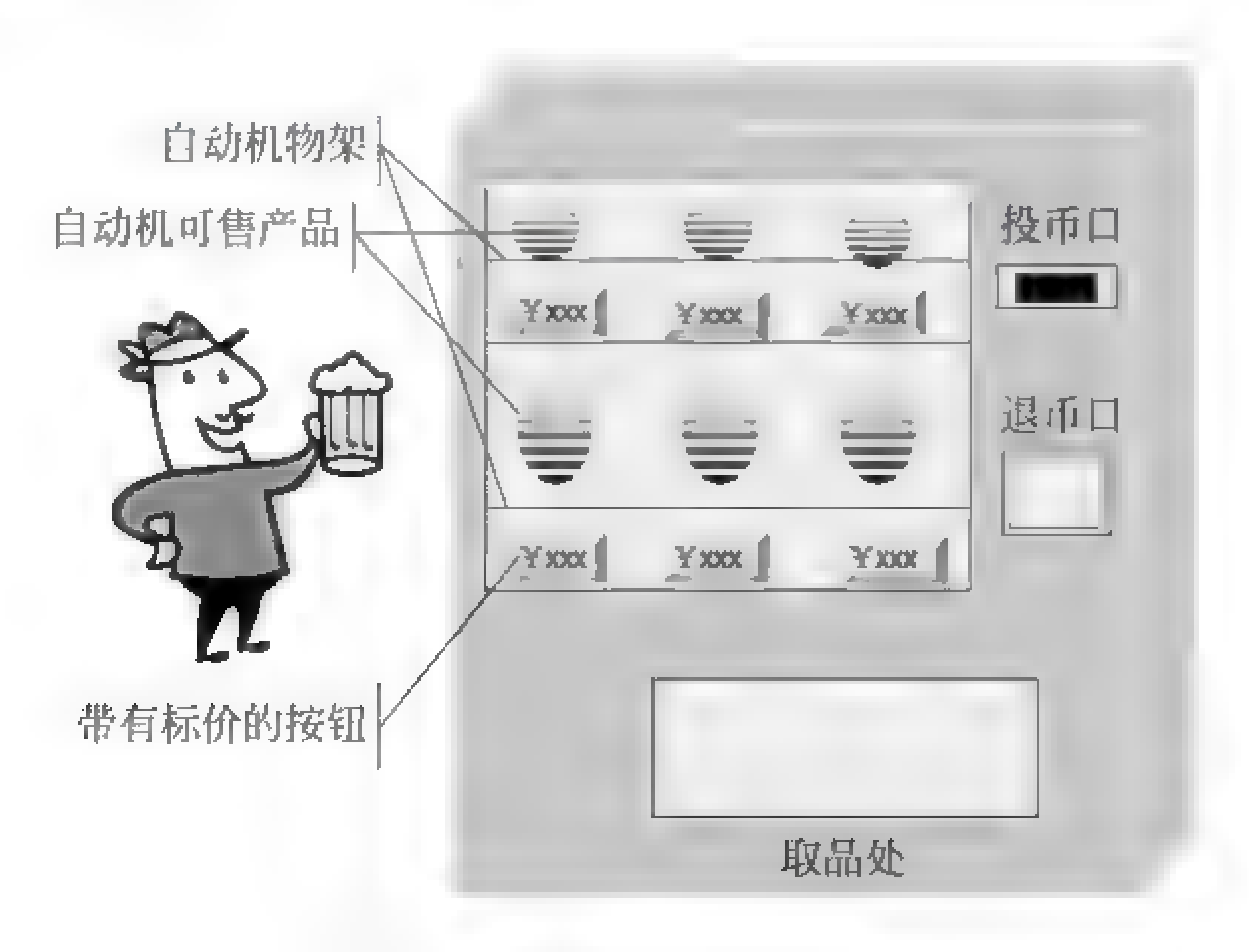


图 6-6 自动售货机硬件构成

自动售货机的各个硬件动作是由嵌入式软件控制的。软件控制系统的类图如图 6 7 所示，VendingMachine 由 CoinBox 和 Dispensor 构成，Controller 使用 VendingMachine 实现对自动售货机的控制。下面以 CoinBox 为例，写 一个 JUnit 测试程序来测试 CoinBox 类的有关行为。

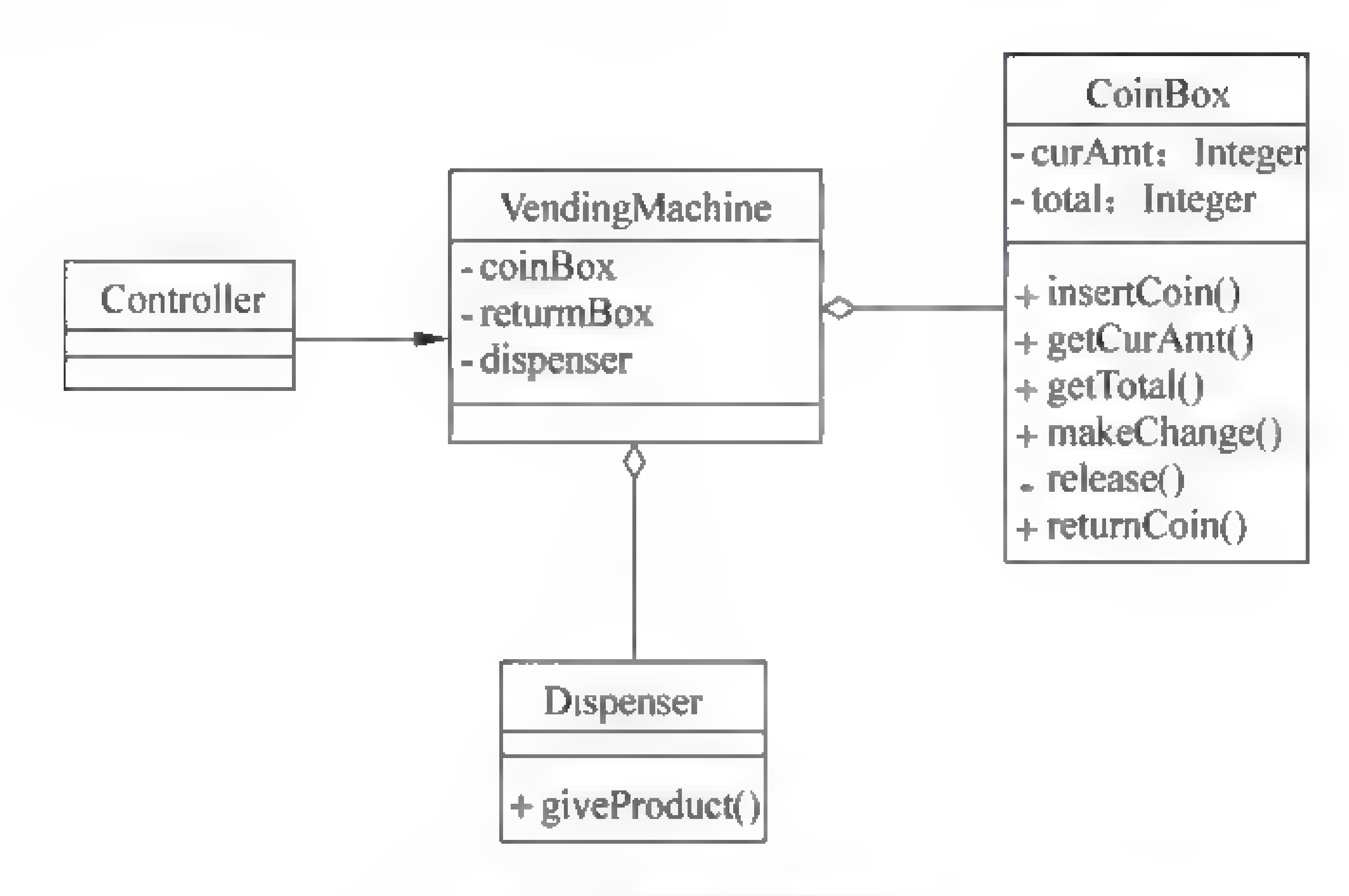


图 6-7 自动售货机类图

清单 6-6 列出了 CoinBox 的一个实现。CoinBox 有两个实例变量(instance variables)：curAmt 和 total；和 6 个方法：insertCoin()、getCurAmt()、getTotal()、makeChange()、release()和 returnCoin()。其中 release()是 private 方法，供 makeChange()和 returnCoin()使用；insertCoin()检查投入币是否是 5 分、10 分或 25 分(币值单位可以理解为美分，有 25 硬币的美分)，如果是，则把币值累加到 curAmt 中；makeChange()找回零钱，对 curAmt 清零，把售出的产品价格累加到 total 中。

清单 6-6：自动售货机 CoinBox 的实现

```
package coinbox;
import java.io.*;
public class CoinBox{
        private int curAmt,total;
        public void insertCoin (int amt) throws InvalidCoinException {
                if (amt!=5&&amt!=10&&amt!=25) {
                        throw new InvalidCoinException();
                }
                this.curAmt += amt;
                System.out.println ("Current Amount is "+curAmt);
        }
        public int getCurAmt () {
                return curAmt;
        }
```

```
        public int getTotal () {
                return total;
        }
        public void makeChange (int price ) throws InvalidPriceException{
                if (price<=0){
                        throw new InvalidPriceException();
                }
                release(curAmt-price);
                curAmt=0;
                total+=price;
        }
        private void release (int amt){
                System.out.println ("Release "+amt);
        }
        public void returnCoin (){
                release (curAmt);
                curAmt=0;
        }
}
```

清单 6-7 列出了上述 CoinBox 的一个 JUnit 测试程序 CoinBoxTest。

(1) 实例变量 coinBox 在 setUp()中被设置为一个 CoinBox 对象的测试置具。

(2) CoinBoxTest 里有唯一的测试方法 testTotal()。在这个测试方法中，调用 coinBox 的 insertCoin()10 次，每次投入 10 分。调用 coinBox 的 getTotal ()，并把获得的值附给局部变量 total。使用 assertTrue("Total=" + total + " Expected=100",total==100)断言来验证"total==100"。注意，这里使用了带有消息的参数断言格式。

清单 6-7：自动售货机 CoinBox 的 JUnit 测试 1

```
import junit.framework.*;
public class CoinBoxTest extends TestCase{
        private CoinBox coinBox;
        public CoinBoxTest(String testCaseName){
                super(testCaseName);
        }
        public void setUp() {
                coinBox=new CoinBox();          ①
        }
        public void testTotal() throws InvalidCoinException{
                for(int i=1; i<=10; i++)
                        coinBox.insertCoin(10);
                int total=coinBox.getTotal();                  ②
                assertTrue("Total ="+total+" Expected =100",total==100);
        }
}
```

利用默认 TestSuite，现在可以把清单 6-7 中的测试程序编译并用 JUnit 的 AWT TestRunner 运行了。请读者在看下面的内容之前想一下，其运行的结果应该是什么情况？

用 JUnit 的 TestRunner 运行的结果如图 6-8 所示，结果显示横条呈红色。统计结果报告：运行一个测试用例，出现一个"失败"。错误与失败上方区里显示失败的测试方法 testTotal(coinbox: CoinBox)，注意"："后面的 Total=0 Expected=100 是带有 String 参数的 assertTrue(("Total=" + total + " Expected=100", total==100)断言输出的消息。这个消息使得失败的原因很清楚：实际结果为 Total=0，而期望结果是 Expected=0。如果再去看一下清单 6-6 列出 CoinBox 的 insertCoin()方法，就容易明白，insertCoin()方法只对实例变量 curAmt 操作，而没有对 total 有任何影响；测试方法 testTotal()只调用 insertCoin()方法，所以当验证 getTotal()时，其返回值是 0。

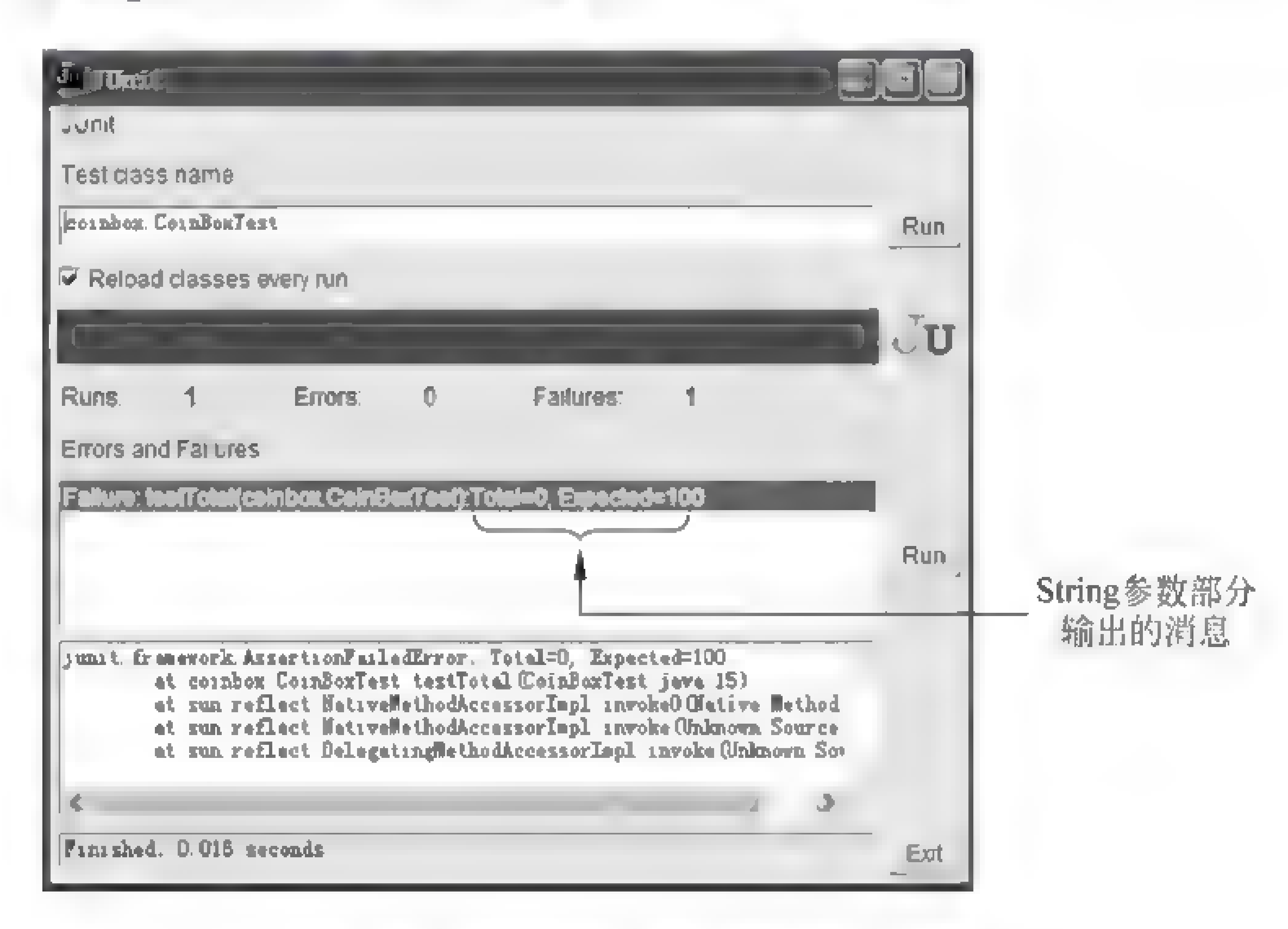


图 6-8 CoinBox 的 JUnit 测试运行结果 1

testTotal()也可以使用 assertEquals 断言，这样在遇到失败时，会自动打印出期望的值与实际的值。

如果把清单 6-7 中的 testToal()方法换成 testCurAmt()，如清单 6-8 所示，运行的结果又是如何呢？testCurAmt()里的 for 语句部分与 testToal()相同，但这次调用 coinBox 的是 getCurAmt()，并把获得的值附给局部变量 curAmt。使用 assertTrue ("CurAmt= "+curAmt+" Expected=100",curAmt ==100)断言来验证"curAmt==100"。

清单 6-8：自动售货机 CoinBox 的 JUnit 测试 2

```
import junit.framework.*;
public class CoinBoxTest extends TestCase{
        private CoinBox coinBox;
```

```
        public CoinBoxTest(String testCaseName){
                super(testCaseName);
        }
        public void setUp() {
                coinBox=new CoinBox();
        }
        public void testCurAmt() throws InvalidCoinException,InvalidPriceException{
                for(int i=1; i<=10; i++)
                        coinBox.insertCoin(10);
                int curAmt=coinBox.getCurAmt();
                assertTrue("curAmt="+curAmt+"Expected=100",curAmt==100);
        }
}
```

编译清单 6-8 中的测试程序并用 JUnit 的 TestRunner 运行的结果如图 6-9 所示。这次既没有失败也没有错误，测试通过。细心的读者可能注意到了，这次断言 assertTrue ("CurAmt="+curAmt+" Expected=100",curAmt ==100)没有输出消息。不错，当无失败时，这种格式的断言不输出消息。



图 6-9　CoinBox 的 JUnit 测试运行结果 2

6.2　JUnit 的设计

本节将基于 JUnit 3.8.x 版本介绍 JUnit 的设计目标和设计核心内容，包括 TestCase、TestSuite 和 TestResult。在学习设计内容时，要用到设计模式(Design Pattern)知识，如果

读者还不具备这方面知识的话，虽然仍能理解 JUnit 思想，但总不能深刻。希望通过这一节的学习，能引起读者对于学习设计模式的兴趣。

6.2.1 设计目标

根据引自 JUnit A Cook's Tour[4]的文章，JUnit 的设计目标主要有 3 条：

(1) 提供一个测试框架，使程序开发人员利用他们熟悉的工具方便地为程序写一个新的测试，并且避免付出重复的努力。

(2) 提供一种管理测试用例的机制，使得测试通过的程序在以后任何时候，可以运行当时的测试用例，来验证那个程序。而且这些测试用例可以和后来的新测试用例组合在一起测试。

(3) 提供一种重用方式，使得测试框架能重新使用测试置具来运行不同的测试。

下面来看如何设计 JUnit 以达到上述 3 条目标。

6.2.2 JUnit 设计

本节介绍的内容包括 3 个类：TestCase、TestSuite 和 TestResult。

1. TestCase

清单 6-9 给出了 TestCase 的一个轮廓。

(1) 在 JUnit 框架中，TestCase 是一个"抽象(Abstract)"类，它的子类通过"继承"来实现重用。implements Test 部分现在暂且不用考虑，在讨论 TestSuite 时再回过头来讲解这部分。在将测试用例封装成对象的时候，使用了 Command 模式(如图 6-10 所示)，它让我们可以为一个操作生成一个对象，并给出一个 execute(这里是 run)方法。

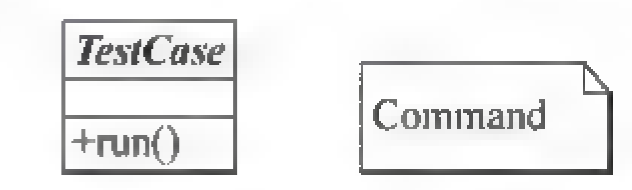


图 6-10 TestCase 中的 Command 模式

(2) 当创建每个 TestCase 时，都有一个名字，以便当某个测试未通过时，用该名字标识这个测试。

(3) 所有的测试都有一个相同的结构，即设置测试置具，用测试置具运行一些代码，检查结果，然后清除测试置具。遵循这样步骤使得每个测试都有一个全新的测试置具，测试结果之间互不影响。每个步骤因测试不同而有些不同，这些不同将留给子类去实现。我们所描述的问题正是"模板方法模式(Template Method Pattern)"所要解决的问题。run (TestResult result)依次调用 setUp()、runTest()和 tearDown()3 个方法就是采用了 Template Method 设计模式，如图 6 11 所示。

(4) 在 JUnit 框架的 TestCase 中，setUp()、runTest()和 tearDown()的默认实现是不

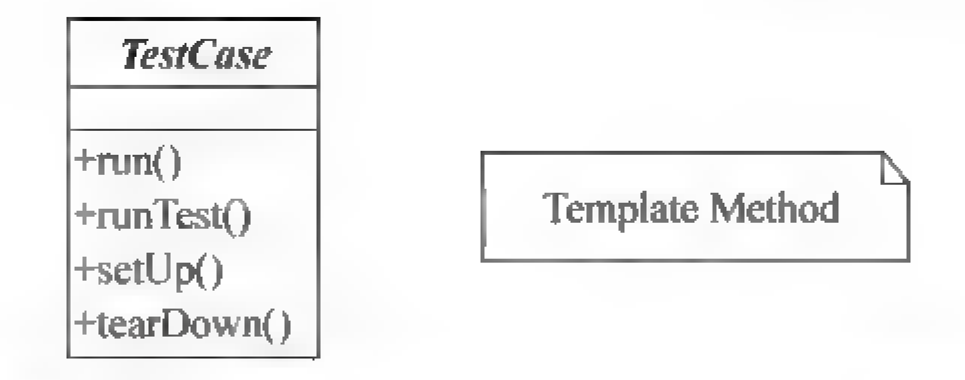


图 6-11 TestCase 中的 Template Method 模式

做任何事情。它们的功能留给子类去实现。

(5) JUnit 的设计者把每个测试用例都看作不同的对象(Object),虽有相似的测试步骤,但各自有不同的测试内容,还有不同的识别名称。JUnit 框架有"命令模式(Command Pattern)"用同一接口来封装(Encapsulate)不同测试对象。在 JUnit 框架中,这个接口是 run()。这使得 JUnit 框架运行测试时只认准一个接口,而开发人员则可以写出不同的任意多个测试用例。

关于 createResult()方法,将在下一节将继续讨论。

清单 6-9:JUnit 的 TestCase 抽象类

```
public abstract class TestCase implements Test {      ①
  private final String fName;  ←──┐
  public TestCase(String name) { ←─┘── ②
    fName= name;
  }
  public void run(TestResult result) {←── ③
    result.startTest(this);           │
    setUp();   ┐                      │
    runTest(); ├ ←────────────────────┘
    tearDown();┘
  }
  protected void runTest() {   ┐
  }                            │
  protected void setUp() {     ├ ④
  }                            │
  protected void tearDown() {  ┘
  }
  public TestResult run() {  ⑤
    TestResult result= createResult();
    run(result);
    return result;
  }
  protected TestResult createResult() {
    return new TestResult();
  }
```

```
    … //省略其他方法
}
```

2. TestResult

对于测试运行的结果需要有一个报告，一般包括这样的信息：运行了多少测试用例，有多少测试用例通过，有多少没有通过，是哪些测试用例没有通过。JUnit 的每一个测试都是通过运行 TestCase 中的 run()方法执行的，因此在 run()中统计各个测试结果是很直接的。

JUnit 框架使用“SmallTalk 最佳实践模式”的“收集参数(Collecting Parameter)”模式。这个模式建议在被收集结果的方法上添一个参数，并传入一个对象，让这个对象收集结果。TestCase 中的 run(TestResult result)方法是采用了收集参数模式。TestResult 的代码轮廓如清单 6-10 所示。

(1) TestResult 是传入 run()的对象，用来收集运行测试的结果。

(2) 用 run(TestResult result)方法调用 TestResult 的 startTest(Test test)(见清单 6-9)，来跟踪记录测试运行的数目。TestResult 的 startTest 方法被声明为 synchrionized，这使得当有多个测试运行在不同的线程上时，单一的 TestResult 的对象能够安全地收集结果，如图 6-12 所示。

清单 6-10：JUnit 的 TestResult 类

```
public class TestResult extends Object {   ①
    protected int fRunTests;
    public TestResult() {
        fRunTests= 0;
    }
    public synchronized void startTest(Test test) {   ②
        fRunTests++;
    }
    … //省略其他方法
}
```

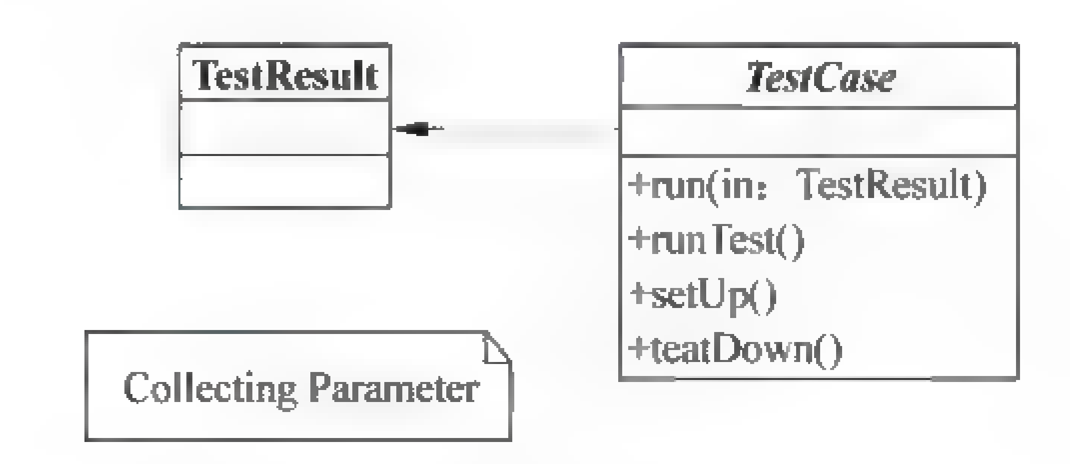


图 6-12 TestResult 中的 Collecting Parameter 模式

再去看一下清单6-9，为了保持TestCase对外的简单接口，TestCase有无参数的run()方法，run()利用createResult()，创建了自己的TestResult对象，并把这个对象传入有参数的run(result)。在run(result)调用TestResult的startTest方法，开始统计工作。

在6.1.3节中，我们曾讨论过失败与错误。失败的可能性是可预期的，并且使用断言来进行检查；而错误是不可预期的，如ArrayIndexOutOfBoundsException异常。下面将结合run(TestResult result)，对JUnit框架的失败与错误做进一步讨论。清单6-11展开了run(TestResult result)的代码：

(1) 失败是通过一个AssertionFailedError来发送通知的。在第一个catch子句中对失败进行捕获。

(2) 在第二个catch子句则捕获其他所有的异常，以确保测试能够继续运行。

清单6-11：TestCase的run(TestResult result)方法

```
public void run(TestResult result) {
    result.startTest(this);
    setUp();
    try {
        runTest();
    }
    catch (AssertionFailedError e) {   ①
        result.addFailure(this,e);
    }
    catch (Throwable e) { ②
        result.addError(this,e);
    }
    finally {
        tearDown();
    }
}
```

注意，第一个catch子句中的内容和第二个catch子句中的内容不能对换位置，否则永远无法捕获到AssertionFailedError。因为AssertionFailedError是继承Java的Error类(见清单6-12)，而Error类是继承Throwable。如果将第二个catch子句中的内容置于第一个catch子句中的内容之前，所有类型的异常，包括AssertionFailedError类型的，都被JUnit的"错误"捕获。清单6-11中的run(TestResult result)保证AssertionFailedError类型的异常，先将其记入"失败"，而其他类型的异常全记入不可预期的错误。这样便体现了在JUnit框架中失败与错误的分别。

从清单6-11中可以看出，AssertionFailedError不应该由客户程序(指TestCase中的一个测试方法)来负责捕获，而应该由Template Method内部的TestCase.run()来负责。

清单 6-12：AssertionFailedError 继承 Java 的 Error 类

```
public class AssertionFailedError extends Error {
        private static final long serialVersionUID=1L;

        public AssertionFailedError () {
        }
        public AssertionFailedError (String message) {
                super (message);
        }
}
```

TestCase 提供的断言方法会触发一个 AssertionFailedError。针对不同的目的 JUnit 提供一组断言方法。下面只是我们已经见过的(见清单 6-3)中最简单的一个：

```
protected void assertTrue(boolean condition) {
    if (! condition)
      throw new AssertionFailedError();
}
```

在 TestResult 中，addError()和 addFailure()方法是用来收集错误的，如下所示：

```
public synchronized void addError(Test test,Throwable t) {
    fErrors.addElement(new TestFailure(test,t));
}

public synchronized void addFailure(Test test,Throwable t) {
    fFailures.addElement(new TestFailure(test,t));
}
```

addError()和 addFailure()用到的 TestFailure 是框架中一个小的内部帮助类(Helper Class)，它将失败的测试和异常绑定在一起以备测试执行完成后报告。

3. TestSuite

为了获得对系统状态的信心，一般需要运行许多测试。前面介绍的 TestCase 和 TestResult，使得 JUnit 能够运行一个单独的测试用例，并在一个 TestResult 中报告结果。接下来的挑战是要对其进行扩展，以使其能够运行许多不同的测试用例。如果测试调用者不必关心其运行的是一个或多个测试用例，那么这个问题便能够轻松解决。在 JUnit 框架中，这个问题是通过使用“组合模式(Composite)”来解决的。

根据《设计模式》一书[5]，组合模式的意图是：“将对象组合成树型结构以表示‘部分-整体’的层次结构。Composite 使得用户对单个对象和组合对象的使用具有一致性。”在这里“部分-整体”的层次结构是让人感兴趣的地方，因为能够支持测试层层相套的套件正是问题的所在。

简要介绍一下组合模式,Composite。用描述模式的语言来说,Composite 引入了如下的参与者。

- Component:声明想要使用的接口。此接口用来与测试进行交互。
- Composite:实现该接口并维护一个测试的集合。
- Leaf:代表集合中的一个测试用例,符合 Component 接口规范。

组合模式告诉我们要引入一个抽象类,来为单独的对象和 Composite 对象定义公共的接口。这个类的基本意图就是定义一个接口。在 Java 中应用组合模式时,更倾向于定义一个接口,而非抽象类。使用接口避免了将 JUnit 托付于一个具体的基类来进行若干测试。所必需的是这些测试要符合这个接口。因此 JUnit 对组合模式的描述进行变通,引入一个 Test 接口,作为 Component 的接口。

```
public interface Test {
    public abstract void run(TestResult result);
}
```

TestCase 对应组合模式中的一个 Leaf。现在回头看一下清单 6-9,当时没有解释 implements Test 部分,现在应该明白了,TestCase 是实现了 Test 接口里的抽象方法 run (TestResult result)。

下面介绍参与者 Composite 如图 6-13 所示。JUnit 框架将其取名为 TestSuit(测试套件)类。清单 6-13 给出了 JUnit 的 TestSuit 类轮廓。

(1) TestSuit 在一个 Vector 中保存了其子测试(Child Test)。

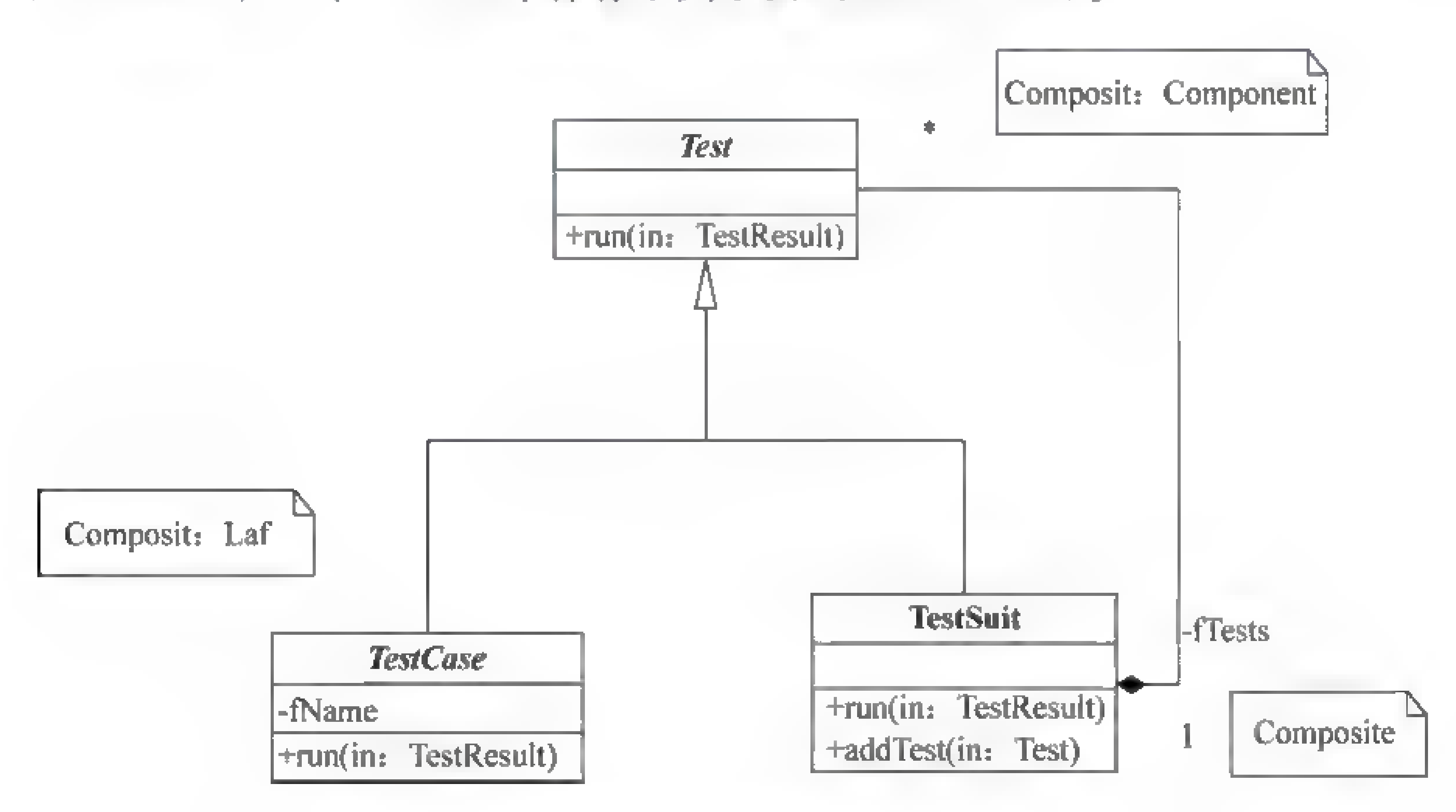


图 6-13 Test、TestCase、TestSuit 中的 Composite 模式

(2) TestSuit 的 run(TestResult result)方法把对其调用委托(Delegate)给子成员来进行。当调用 TestSuite 对象的 run(TestResult result) 方法,TestSuit 遍历自己容纳的所有

子测试，逐个调用每个测试的 test. run(result)。

(3) JUnit 框架的客户必须能将测试添加到一个套件中，这是使用 TestSuit 的 addTest 方法来实现的。

清单 6-13：JUnit 的 TestSuit 类

```
public class TestSuite implements Test {            ①
    private Vector fTests=new Vector();
    public void run(TestResult result) {            ②
        for(Enumeration e=fTests.elements(); e.hasMoreElements(); ){
            Test test=(Test)e.nextElement();
            test.run(result);
        }
    }
    public void addTest(Test test) {                ③
        fTests.addElement(test);
    }
    … //省略其他方法
}
```

注意，所有上面的代码是仅依赖于 Test 接口的。由于 TestCase 和 TestSuit 两者都符合 Test 接口，所以可以递归地将测试套件再组合成套件。开发者能够创建他们自己的 TestSuit，还可创建一个套件来组合这些 TestSuit 并运行。

6.2.3 小结

前面介绍了 JUnit 框架里主要类的设计，包括 TestCase、TestResult 和 TestSuite。请读者思考一下，JUnit 框架的设计是如何来实现其目标的？

6.2.1 节列出了 3 个设计目标。第二个设计目标直接与 TestSuite 关联。TestSuite 提供了一种管理测试用例的机制，使得测试通过的程序在以后任何时候都可以运行当时的测试用例，以来验证程序。而且这些测试用例可以和后来的新开发的测试用例组合在一起进行测试。这种支持修复或更正后的“再测试”其实是单元回归测试，它有利于提升系统的可信赖度。关于回归测试的有关知识，在第 7 章有深入的讨论。

第三个设计目标是由 JUnit 里的测试置具来支持的。测试置具是运行一个或多个测试所需的公用资源或数据集合。TestCase 通过 setUp 和 tearDown 方法来自动创建和销毁测试置具。TestCase 会在运行每个测试之前调用 setUp，并且在每个测试完成之后调用 tearDown。把不止一个测试方法放入同一个 TestCase 的一个重要理由就是可以共享测试置具代码。测试置具提供一种重用方式，使得测试框架能重新使用测试置具来运行不同的测试。

第二个设计目标和第三个设计目标是支持第一个设计目标的。JUnit 框架里的

TestCase、TestResult、TestSuite 还有 TestRunner(我们没有介绍)一起,提供一个测试框架,使得程序开发人员利用他们熟悉的工具,如 Java 等,很方便地为程序编写一个新的测试,并且避免付出重复的努力。

作为 JUnit 总结附录,下面列出 JUnit 框架中的主要接口和类,包括 Test 接口、运行测试和收集测试结果等。如图 6-14 所示。

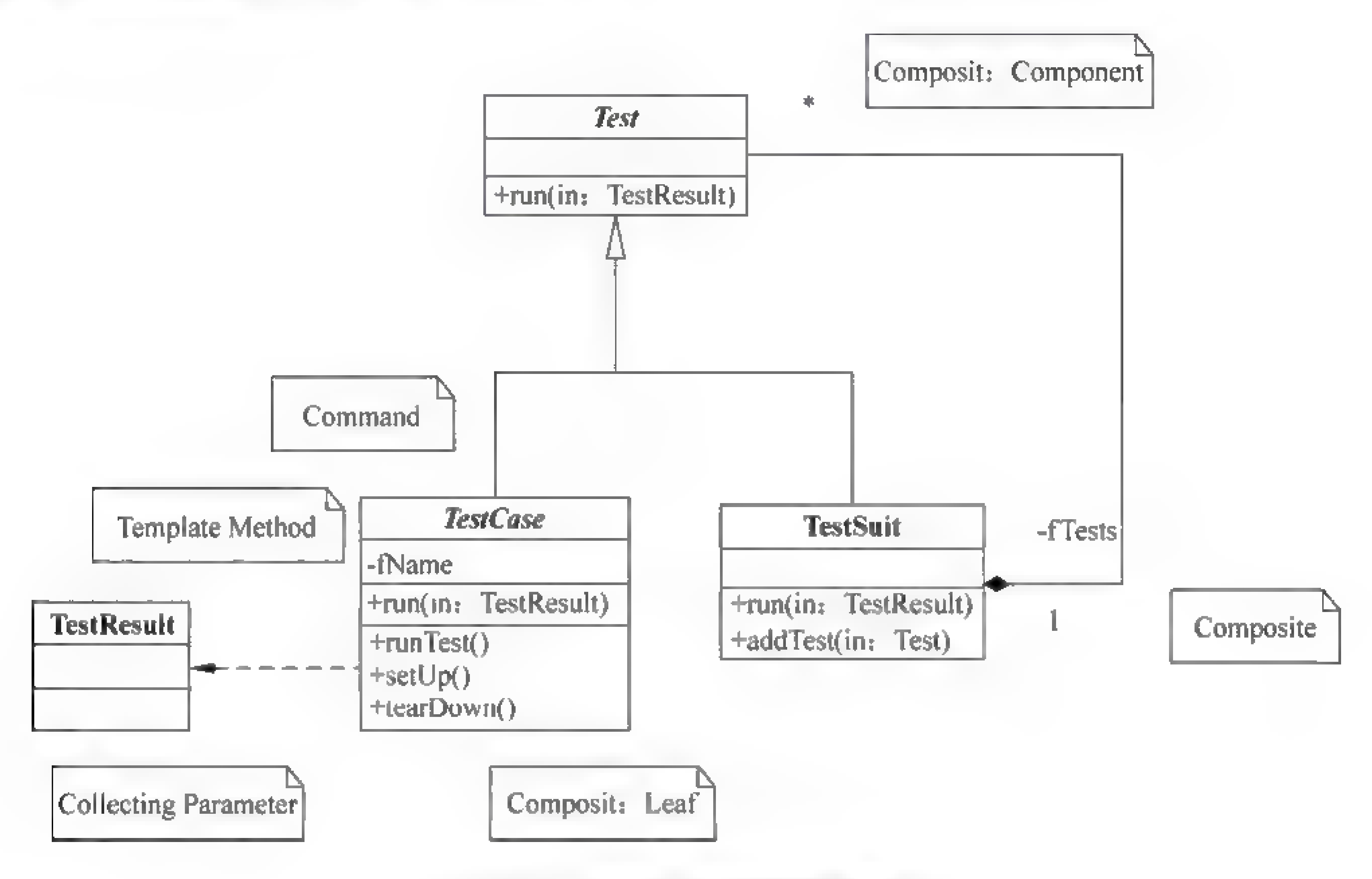


图 6-14 JUnit 中使用的设计模式

1. Test 接口

使用了 Composite 设计模式,是单独测试用例(TestCase)、聚合测试模式(TestSuite)及测试扩展(TestDecorator)的共同接口。它的 public int countTestCases()方法,用来统计这次测试有多少个 TestCase,另外一个方法就是 public void run(TestResult),TestResult 实例接受测试结果,run 方法执行本次测试。

2. TestCase 抽象类——定义测试中的固定方法

TestCase 是 Test 接口的抽象实现,(不能被实例化,只能被继承)其构造函数 TestCase(string name)根据输入的测试名称 name 创建一个测试实例。由于每一个 TestCase 在创建时都要有一个名称,所以若某测试失败了,便可识别出是哪个测试失败。

TestCase 类中包含 setUp()、tearDown()方法。setUp()方法集中初始化测试所需的所有变量和实例,并且在依次调用测试类中的每个测试方法之前再次执行 setUp()方法。tearDown()方法则是在每个测试方法之后,释放测试程序方法中引用的变量和实例。这

两个方法的实现不是必需的。开发人员编写测试用例时，只需继承 TestCase，来完成 run 方法即可，然后 JUnit 获得测试用例，执行它的 run 方法，把测试结果记录在 TestResult 之中。

3. Assert 静态类——一系列断言方法的集合

Assert 包含了一组静态的测试方法，用于比对期望值和实际值，如果比对不正确，即测试失败，Assert 类就会抛出一个 AssertionFailedError 异常，JUnit 测试框架将这种错误归入 Failes 并加以记录，同时标记为未通过测试。如果该类方法中指定一个 String 类型的参数，则该参数将被作为 AssertionFailedError 异常的标识信息，告诉测试人员修改异常的详细信息。

4. TestSuite 测试包类——多个测试的组合

TestSuite 类负责组装多个测试用例。待测的类中可能包括了对被测类的多个测试，而 TestSuit 负责收集这些测试，使我们可以在一个测试中完成全部的对被测类的多个测试。TestSuite 类实现了 Test 接口，且可以包含其他的 TestSuites。它可以处理加入 Test 时的所有抛出的异常。

TestSuite 处理测试用例有 6 个规约(否则会被拒绝执行测试)：

- 测试用例必须是公有类(Public)。
- 测试用例必须继承自 TestCase 类。
- 测试用例的测试方法必须是公有的(Public)。
- 测试用例的测试方法必须被声明为 Void。
- 测试用例中测试方法的前置名词必须是 test。
- 测试用例中测试方法误任何传递参数。

5. TestResult 结果类和其他类与接口

TestResult 结果类集合了任意测试的累加结果，通过 TestResult 实例传递个每个测试的 Run()方法。TestResult 在执行 TestCase 时如果失败，则会抛出异常。TestListener 接口是个事件监听规约，可供 TestRunner 类使用。它通知监听器对象相关事件，方法包括测试开始 startTest(Test test)、测试结束 endTest(Test test)、增加异常 addError(Test test, Throwablet)和增加失败 addFailure(Test test, AssertionFailedError t)。

TestFailure 失败类是个“失败”状况的收集类，解释每次测试执行过程中出现的异常情况。其 toString()方法返回“失败”状况的简要描述。

JUnit 的优秀设计，使得可以很方便地对其进行扩展，很多流行的测试工具都是基于 JUnit 的，例如 StrutsTestCase 和后面提到的 DBUnit。

6.3 模仿对象测试

单元测试已作为软件开发的最佳实践被普遍接受。当编写某个类时,需要另外提供一个相应的测试类,用各种参数调用被测类的各种公用方法,通过验证返回值是否正确来确保被测类实现的正确性。对于逻辑简单或自身相对独立的类来说,编写单元测试很简单。回想第3章的3.1节介绍JUnit时使用的例子,都属于这种情况。为了使读者把注意力集中在JUnit的使用上,例子中被测类中的方法都很简单,并且没有使用其他类的对象方法来完成其功能。然而,实际系统中可能会有复杂的体系结构,例如某些类会使用底层基础类提供的功能或服务。这些被使用的类称为合作者(Collaborator)。当对这种复杂类进行单元测试时,实例化这些合作者通常是昂贵的、不切实际的或低效率的。最常见的情况就是合作者类还没有实现,无法使用它提供的功能,而又需要对使用合作者类的类进行单元测试,这时怎么办呢?这就需要使用下面将要介绍的模仿对象技术。

6.3.1 模仿对象简介

隔离其他方法或环境而对被测类中的方法进行单元测试,显然,这是个值得追求的目标。隔离测试可以测试还未完成的代码,只要有接口可用便能进行。最大的好处在于编写专门测试单一方法的代码,免去了被测试方法调用其他对象而带来的副作用。编写小而专一的测试很有用:小的测试容易理解,当代码的其他部分改变时测试也不易被破坏。如果测试的粒度比较大,那么一旦重构引入错误,就会有一系列的测试失败。结果就是测试报告出现了错误,但并不能确切知道错误在哪儿。而用细粒度的测试,受错误影响的测试比较少,而且测试会提供精确信息指出错误的确切原因。

模仿对象(Mock Object)就是实现隔离测试的一种技术。模仿对象用来代替被测代码中的合作者对象。被测试对象调用"模仿"对象,而不是调用"真实"对象。模仿对象会传递回结果,结果是设置好的。如何创建模仿对象呢?请看下面一个简单的例子。

如图6 15所示,Purchase类是要测试的目标类。Purchase类的getTotalPrice方法使用DBAccess类的getPriceFromDB方法。DBAccess类是Purchase类的合作者。TesterPurchase类使用真实的DBAccess合作者对象对Purchase类进行单元测试。

清单6 14给出了被测类Purchase中getTotalPricie方法的实现(其他的方法没有展示出)。getTotalPricie的实现很简单,它接受传入的一个DBAccess实例dbAccess,然后调用dbAccess的getPriceFromDB方法。

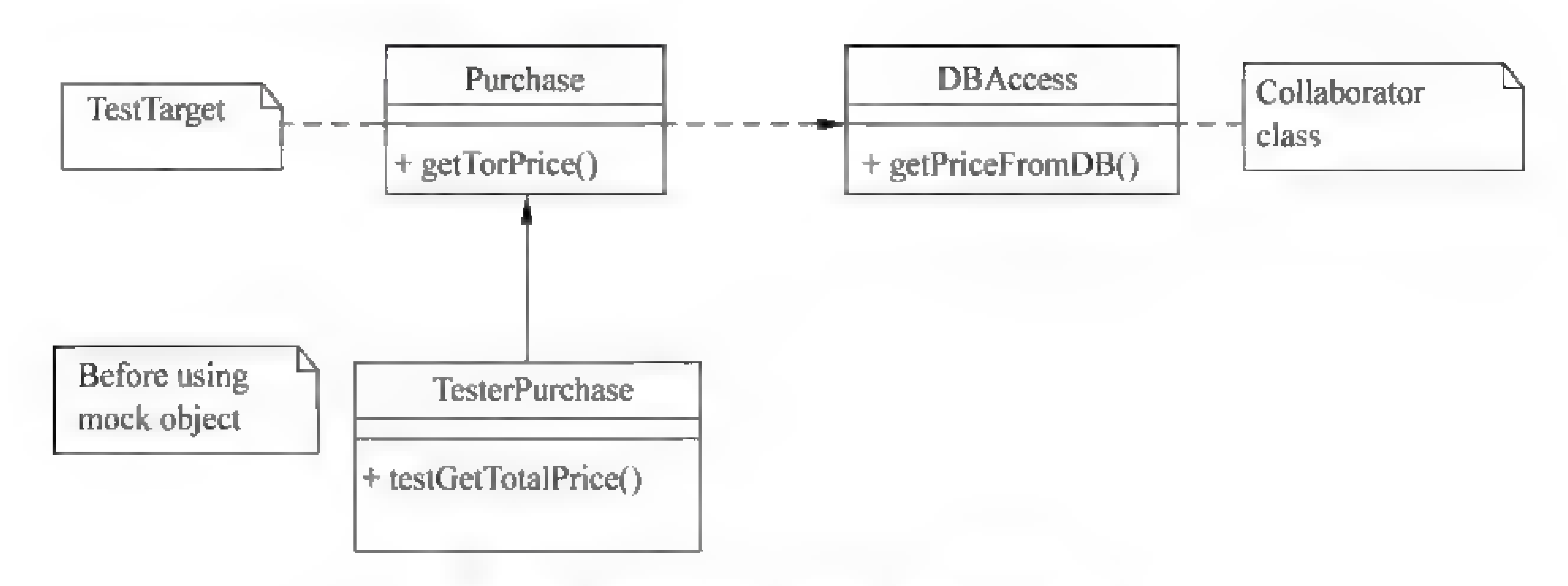


图 6-15 使用 Mock Object 前的单元测试

清单 6-14：被测类 Purchase(只展示部分实现)

```
class Purchase {
...
  public double getTotalPrice(DBAccess dbAccess) {
    dbAccess.getPriceFromDB();
  }
...
}
```

假设 DBAccess 类的实现已完成并通过单元测试，设置数据库里的 Price 值为 180。用"真实"的 DBAccess 来测试 Purchase，如清单 6-15 所示。

清单 6-15：用"真实"的 DBAccess 来测 Purchase

```
class PurchaseWithRealDBAccessTest extends TestCase {

 public void testGetTotalPrice {
  Purchase purchase = new Purchase();
  DBAccess dbAccess = new DBAccess();
  assertTrue(180.0, purchase.getTotalPrice(dbAccess));
 }
}
```

使用"真实"的对象

但是如果 DBAccess 类的实现还未完成，或是数据库的设置不能完全被测试人员控制，导致所设值发生改变，这些情况会使"自动测试"难以达到目的。此时，模仿对象提供了一个解决方案：模仿对象有与被测对象的合作者完全一致的接口，在测试中作为"合作者"被传递给被测对象；当被测对象调用合作者时，模仿对象根据测试者的意愿改变某些状态或返回期望的结果，以检查被测程序是否按照所期望的逻辑进行工作，达到单元测试的目的。

如图 6-16 所示，MockDBAccess 是模仿对象，它继承自真实的 DBAccess，并重写 getPriceFromDB 方法。测试 Purchase 类时，不是调用 DBAccess 的 getPriceFromDB 方法，

而是调用 MockDBAccess 的 getPriceFromDB 方法。

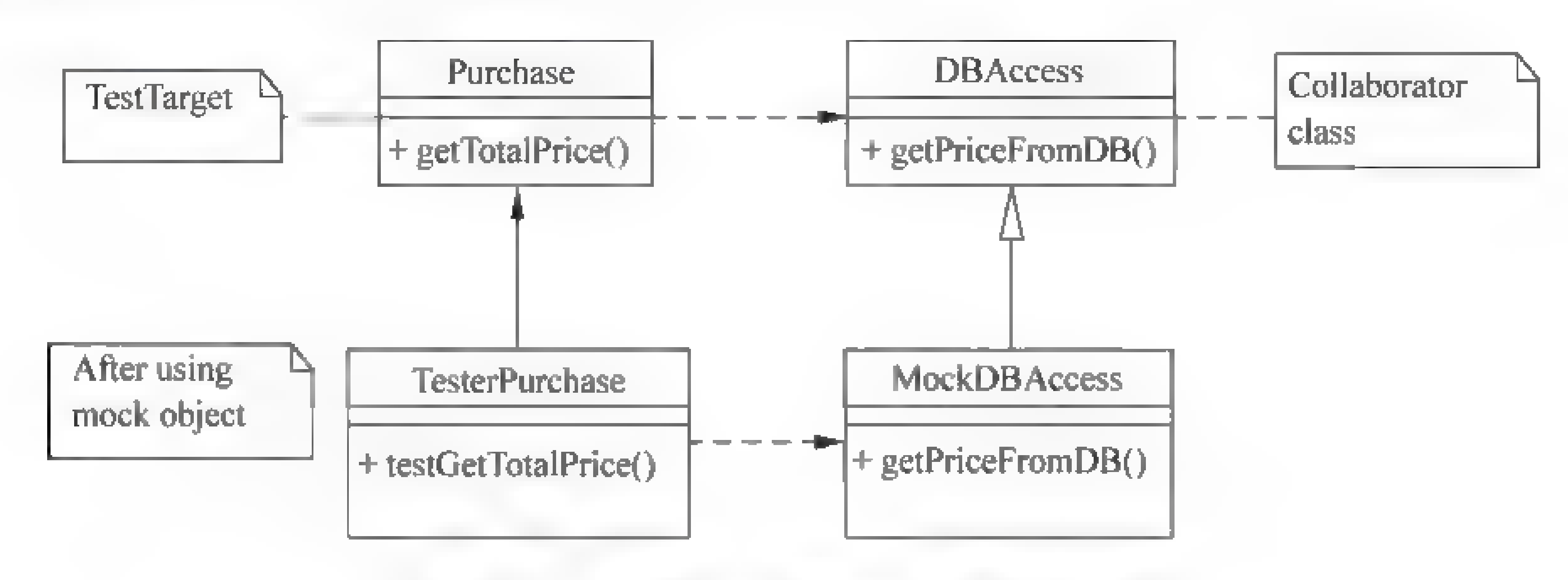


图 6-16 使用 Mock Object 后的单元测试

清单 6-16 是 MockDBAccess 的实现，它只有一个 getPriceFromDB 方法，将 Price 设为 180.0，而不是从数据库获取，并简单地返回 double 类型，值为 180.0。

清单 6-16：DBAccess 的 Mock Object：MockDBAccess

```
class MockDBAccess extends DBAccess {
 public double getPriceFromDB() {
  double price=180.0;
  return price;
 }
}
```

清单 6-17 列出用模仿对象 MockDBAccess 对被测类 Purchase 的 getTotalPrice 进行测试，传入的是 MockDBAccess 实例，而不是如清单 6-16 所示，传入的是 DBAccess 实例。

清单 6-17：利用 MockDBAccess 测试 Purchase

```
class PurchaseWithMockDBAccessTest extends TestCase {

 public void testGetTotalPrice {
  Purchase purchase=new Purchase();
  MockDBAccess mockDBAccess=new MockDBAccess();
  assertTrue(180,purchase.getTotalPrice(mockDBAccess));
 }
}
```

用模仿对象代替“真实”对象

6.3.2 模仿对象与重构

使用模仿对象进行测试的一般编码格式是：

- 创建模仿对象的实例。

- 设置模仿对象中的状态和期望值。
- 将模仿对象作为参数来调用被测代码。
- 验证模仿对象中的一致性和被测对象的返回值或状态。

虽然这种模式对于许多情况都非常有效，但模仿对象有时不能被传递到被测对象中。相反，被测对象是用来创建、查找或获得其合作者的。例如，测试对象可能需要获得对Enterprise JavaBean(EJB)组件或远程对象的引用。或者，被测对象会使用具有副作用的对象，如使用File对象，而File对象要删除一些文件。在单元测试中，我们不希望有这些副作用，而希望使用一个没有副作用的模仿对象。这种情形下可以尝试重构对象，使之更便于测试。例如，可以更改方法签名，以便传入合作者对象。

有人认为，单元测试应该对测试中的代码透明：不应该为了简化测试而改变被测代码。这是错误的，单元测试是对被测代码的最好运用。如果被测代码不能很方便地在测试中使用，那么就需要考虑重构代码。

下面用图6-17所示的例子来描述重构代码过程。ATMService类模拟自动取款(ATM)，提供取款(Withdraw)、存款(Deposit)、转账(Transfer)、查询(Inquiry)服务，它需要使用一个数据连接对象connection。数据连接对象为DBDataConnetion对象，实现IDataConnetion接口，需要连接数据库获取账户信息(getAccount)和更新账户信息(UpdateAccount)。账户信息由类AccountInfo表示。

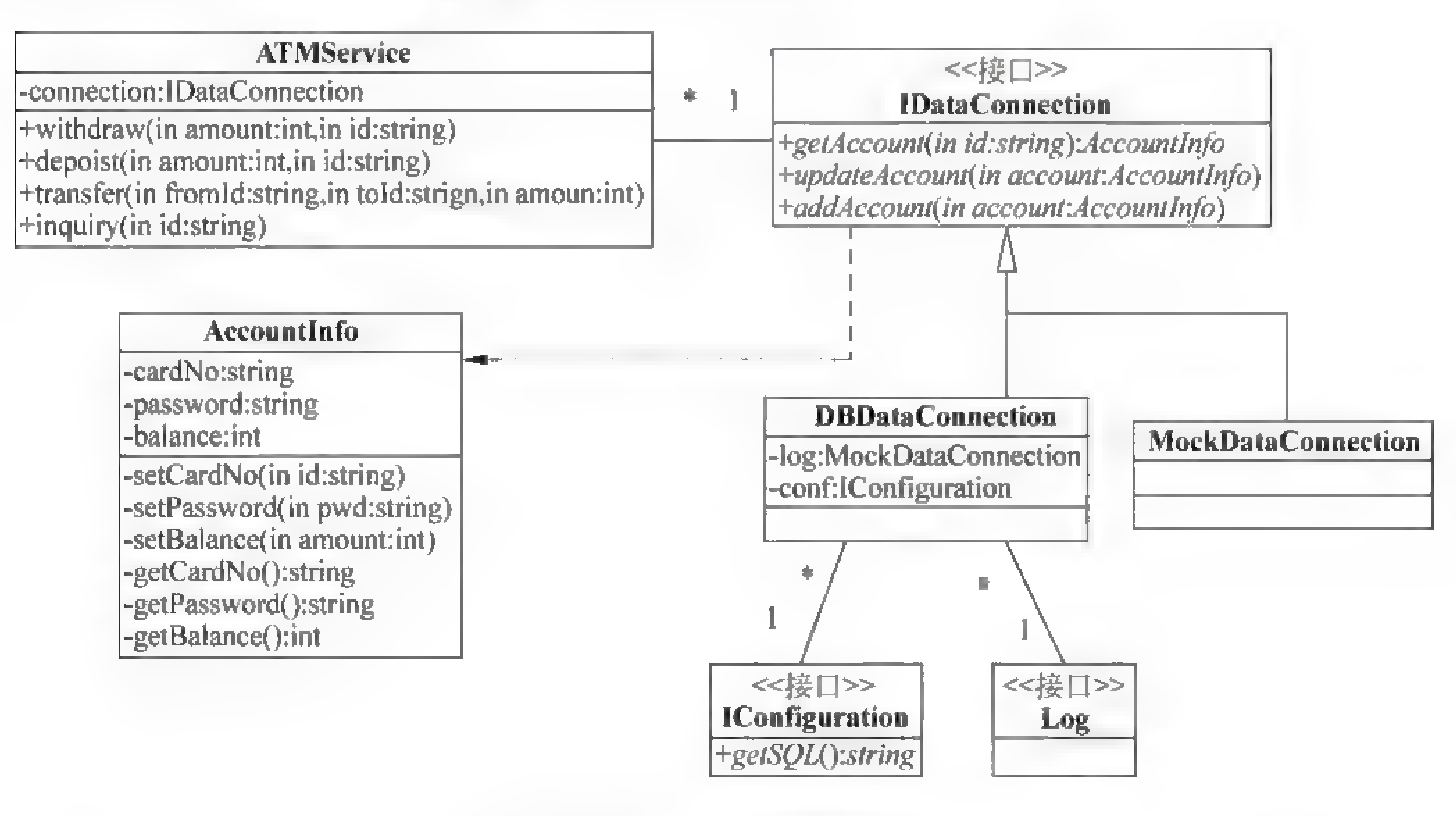


图6-17 ATM实例类图

假设已经实现了ATMService类和AccountInfo类，准备对ATMService类进行单元测试(AccountInfo类只是实现设置和获取属性值，其逻辑非常简单，所以忽略对它的单元测试)。再回顾一下上一节介绍的模仿对象方法。ATMService类中需要使用一个数据连

接对象，但现在还未实现 DBDataConnetion 类，所以需要模仿一个数据连接对象。上一节的例子中直接继承被模仿类得到模仿对象，而本例中有一个提供数据连接功能的接口 IDataConnetion，所以可以通过实现该接口来得到模仿对象 MockDataConnection。

清单 6-18 给出了 ATMService 类的主要代码。ATMService 类使用一个实现了 IDataConnection 接口的数据连接对象，并在构造函数中初始化该对象。有 4 个方法，分别是 withdraw、deposit、transfer、inquiry。

- inquiry(String cardNo)方法传入账户卡号 cardNo，返回该账户的余额。
- withdrawl(String cardNo，long amount)方法传入账户卡号 cardNo 和一定数额的款项 amount，首先通过数据连接对象获得该账户的余额信息，然后从余额中减去 amount，随后再通过数据连接对象更新该账户余额。
- deposit(String cardNo，long amount)方法和 withdrawl 方法类似，只是在账户余额上增加一定数额(amount)。
- transfer(String fromCardNo，String toCardNo，long amount)方法涉及两个账户，从一个账户 fromCardNo 中扣除一定数额 amount 增加到另一个账户 toCardNo 上。相当于 withdrawl 和 deposit 方法的结合。

清单 6-18：被测类 ATMService

```
public class ATMService {
    private IDataConnection dconn;

    public ATMService(IDataConnection dc) {
        dconn = dc;
    }

    public void withdraw(String cardNo,long amount) {
        Account a = dconn.getAccount(cardNo);
        long balance = a.getBalance() -amount;
        a.setBalance(balance);
        dconn.updateAccount(a);
    }

    public void deposit(String cardNo,long amount) {
        Account a = dconn.getAccount(cardNo);
        long balance = a.getBalance() + amount;
        a.setBalance(balance);
        dconn.updateAccount(a);
    }
    public void transfer(String fromCardNo,String toCardNo,long amount) {
        Account a = dconn.getAccount(fromCardNo);
        Account b = dconn.getAccount(toCardNo);
```

```
            long aBalance=a.getBalance()-amount;
            long bBalance=b.getBalance()+amount;
            a.setBalance(aBalance);
            b.setBalance(bBalance);
            dconn.updateAccount(a);
            dconn.updateAccount(b);
        }

        public long inquiry(String cardNo) {
            Account a = dconn.getAccount(cardNo);
            return a.getBalance();
        }
    }
```

清单 6-19 给出 MockDataConnection 的主要代码。MockDataConnection 中用 HashMap 存储账户信息，并在构造函数中预先设好两个账户信息(a 和 b)作为测试数据。账户卡号作为关键字(Key)，账户对象作为值(Value)。通过简单调用 HashMap 的 get 和 put 方法实现 IDataConnection 接口的 getAccount、addAccount 和 updateAccount 方法。

清单 6-19：MockDataConnection

```
public class MockDataConnection implements IDataConnection{
    private HashMap<String,Account> mdb;

    public MockDataConnection() {
        mdb=new HashMap<String,Account>();
        Account a=new Account("1","111111",300);
        Account b=new Account("2","222222",1000);
        mdb.put(a.getCardNo(),a);
        mdb.put(b.getCardNo(),b);
    }

    public Account getAccount(String cardNo) {
        return (Account)mdb.get(cardNo);
    }

    public void updateAccount(Account a) {
        addAccount(a);
    }

    public void addAccount(Account a) {
        mdb.put(a.getCardNo(),a);
    }
}
```

清单6 20列出了用模仿对象MockDataConnection测试ATMService类的代码。把MockDataConnection对象mdc作为构造函数的参数转递给ATMService,这样在ATMService对象atm中使用的就是模仿对象。在每个测试方法中(testXXX),分别调用atm的相应方法,然后通过mdc检查调用后的结果是否正确。记住,在mdc中已经设置好了测试数据。

清单6-20:用MockDataConnection测试ATMService

```
import junit.framework.TestCase;
public class TestATMService extends TestCase {
    private MockDataConnection mdc=new MockDataConnection();
    private ATMService atm=new ATMService(mdc);

    public void testWithdrawl() {
        atm.withdrawl("1",20);
        Account a=(Account)mdc.getAccount("1");
        assertEquals(280,a.getBalance());
    }

    public void testDeposit() {
        atm.deposit("1",20);
        Account a=(Account)mdc.getAccount("1");
        assertEquals(320,a.getBalance());
    }

    public void testTranfer() {
        atm.transfer("2","1",200);
        Account a=(Account)mdc.getAccount("1");
        Account b=(Account)mdc.getAccount("2");
        assertEquals(500,a.getBalance());
        assertEquals(800,b.getBalance());
    }

    public void testInquiry() {
        assertEquals(1000,atm.inquiry("2"));
    }
}
```

接下来实现真正的DBDataConnetion类,它需要连接到数据库,从中取出账户信息或更新账户信息。注意,DBDataConnetion类里使用了两个合作者对象:Log对象log用于日志,ResourceBuddle对象buddle用来获取SQL语句。

清单6 21给出了DBDataConnetion类部分代码,说明如何在DBDataConnetion类中使用两个合作者对象。日志对象log实现为静态类变量,所有的DBDataConnetion对象共用一个Log对象。getConnection方法用来从配置文件中(参数filename)读取数据库信息,连

接数据库并返回数据库连接。

清单中只给出了getAccount方法的实现，其他方法类似。下面解释一下getAccount方法的实现：

(1) ResourceBuddle对象bundle从sql.properties文件中读入写好的sql语句。sql.properties文件的格式是key=value这样的语句集合，例如SELECT=SELECT * FROM account where cardNo=?。通过bundle.getString("SELECT")就可以获得等号右面的sql语句。

(2) 日志对象log的作用就是在每个操作前记录一些信息，便于跟踪程序的运行过程和查找错误。

(3) 数据库操作就是按照JDBC的使用步骤：

① 获得数据库连接(getConnection)。

② 得到sql语句(PreparedStatement)。

③ 设置PreparedStatement中的参数。

④ 执行sql语句。

⑤ 获得执行结果。

清单6-21：DBDataConnetion类部分代码

```
import java.io.*;
import java.sql.*;
import java.util.*;
import org.apache.commons.logging.*;
public class DBConnection implements IDataConnection {
    private static final Log log = LogFactory.getLog(DBConnection.class);

    protected Connection getConnection(String filename) throws Exception { //获取数据库连接
        ...
    }

    public Account getAccount(String cardNo) {
        String cn="";
        String pwd="";
        long b=0;
        Connection conn=null;
        ResultSet rs=null;
        ResourceBundle bundle=PropertyResourceBundle.getBundle("sql");
        try {
            log.info("get database connection");
            conn=getConnection("database.properties");
            log.info("get sql statement");
            String selectStm=bundle.getString("SELECT");
```

```
            PreparedStatement st=conn.prepareStatement(selectStm);
            st.setString(1,cardNo);
            log.info("get result set");
            rs=st.executeQuery();
            rs.next();
            cn=rs.getString("cardNo");
            pwd=rs.getString("pwd");
            b=rs.getLong("balance");
        } catch (Exception e) {
            log.error(e.getMessage());
        }
        finally {
            ...
        }
        log.info("Get account whose card No. is " +cardNo);
        return new Account(cn,pwd,b);
    }
...
}
```

从这段代码中，可以看到两个问题：

(1) 无法使用一个不同的日志对象，因为这是在类内部创建的。例如在测试中，可能想要一个不做任何事的日志对象，但是无法做到。一般来说，像这样的类应该能够使用任何给定的日志对象。这个类的目标不是构造日志对象，而是执行某些数据库操作。

(2) getAccount 方法中使用的 ResourceBundle 对象看上去似乎不错，但是如果以后想要使用 XML 文件配置会怎样呢？在测试中如果不想用配置文件又如何写单元测试呢？这时就遇到了本节开头提到的问题，我们无法传入模仿的合作者对象以进行单元测试。这时就要考虑重构原有代码。

清单 6-22 给出了一种重构方式。提取出 Log 和 IConfiguration 两个接口，如图 6-17 所示。Log 接口实现日志记录功能，IConfiguration 接口实现获取 SQL 语句的功能。通过实现这两个接口分别创建两者的模仿对象，再传入 DBConnection 的构造函数来完成对 DBConnection 的单元测试。这里不再展示 Log 和 IConfiguration 接口模仿对象的实现及对 DBConnection 的单元测试，读者可参考前面的模仿对象例子自己完成。

清单 6-22：重构后的 DBDataConnetion 类

```
public class DBConnection implements IDataConnection {
    private Log log;
    private IConfiguration conf;
    public DBConnection (Log l,IConfiguration c) {
        log=l;
        conf=c;
```

```
        }

        public DBConnection () {
            this(LogFactory.getLog(DBConnectionRefactored.class),
                    new PorpertyConfiguration("sql"));
        }
    ...
    }
```

重构以后，就可以完全从测试代码外围控制记录和配置行为。这样，原有代码更加灵活，可以用任何记录和配置实现。有时这也称为"依赖注入"(Dependency Injection)。需要注意的是，如果先编写测试，就会自觉把代码设计得灵活些。灵活性是单元测试的关键。也就是说，如果测试先行，就不会发生为灵活性而重构代码所需的开销。

6.3.3 利用工具建立模仿对象

建立模拟对象的目的是创建一个轻量级的、可控制的对象来代替测试中需要的真实对象，模拟真实对象的行为和功能，方便单元测试。模仿对象的工具 JMock 和 EasyMock 就是这种机制的实现，使用这些工具可以快速创建模仿对象，定义交互过程中的约束条件等。使用工具创建模仿对象，首先要下载相关的 jar 包，然后在自己的测试类中导入相关的工具类才能使用。

JMock 最新的稳定版本是 JMock2.0。下面的例子展示了 JMock2.0 的基本用法。要熟练掌握 JMock2.0 的完整用法，请参考其官方网站 www.jmock.org。被测类仍使用上一小节的 ATMService 类，这里只测其中的 transfer 方法。读者可以比较一下手工创建模仿对象和使用工具创建模仿对象的不同。

清单 6-23：利用 JMock2.0 测试 ATMService 类 transfer 方法

```
import org.jmock.integration.junit3.*;
import org.jmock.*;
public class TestATMServiceWithJMock2 extends MockObjectTestCase{
    public void testTransfer(){
        //创建 mock object
        final IDataConnection dc = mock(IDataConnection.class);
        //将 mock 对象传入待测对象
        ATMService atm = new ATMService(dc);
        final Account a = new Account("1","111111",500);
        final Account b = new Account("2","222222",1000);
         // 设置期望值
        checking(new Expectations() {{
            //设置 getAccount 的参数和对应返回期望值
```

```
            one (dc). getAccount("1"); will(returnValue(a));
            one (dc). getAccount("2"); will(returnValue(b));
            //设置 updateAccount 的参数,无返回值
             one (dc). updateAccount(a);
             one (dc). updateAccount(b);
    }});
    //执行被测方法
    atm. transfer("1","2",105);
    //验证执行被测方法后的状态
    assertEquals(a. getBalance(),395);
    assertEquals(b. getBalance(),1105);
  }
}
```

EasyMock 最新的版本是 EasyMock2.2。下面的例子简单地展示了 EasyMock2.2 的用法。EasyMock 的 Mock 机制基于状态,首先是录制状态,记录下来待测的方法和参数,返回值等,然后切换为回放状态。而 jMock 没有切换这一步,直接将参数返回值用一句话写出来。读者要熟练掌握 EasyMock2.2 的用法,请参考其官方网站 http://www.easymock.org。

清单 6-24:利用 EasyMock2.2 测试 ATMService 类 transfer 方法

```
import static org. easymock. EasyMock. * ;
import junit. framework. * ; ;
public class TestATMServiceWithEMockTwo extends TestCase{
  private ATMService atm;
  private IDataConnection mock;
  protected void setUp() {
    mock=createMock(IDataConnection. class); // 创建 mock 对象
    atm=new ATMService(mock); //将 mock 对象传入待测对象
  }

  public void testTranfer() {
        Account a = new Account("1","111111",500);
        Account b = new Account("2","222222",1000);
        // 设置期望值
        expect(mock. getAccount("1")). andReturn(a);
        mock. getAccount("2");
        expectLastCall(). andReturn(b);
        mock. updateAccount(a);
        mock. updateAccount(b);
        replay(mock);       //回放
        //执行被测方法
        atm. transfer("2","1",150);
        //验证执行被测方法后的状态
        assertEquals(650,a. getBalance());
```

```
            assertEquals(850,b.getBalance());
            verify(mock);
        }
    }
```

6.3.4 小结

模仿对象用来代替被测代码中的合作者对象，模拟真实合作者对象的行为，它使我们的关注点集中在被测代码上。模仿对象应该尽量简单，不包含任何业务逻辑，只做测试要求它做的事情。大多数情况下，编写模仿对象有一个正面的副作用：它迫使你重写部分被测试的代码。实践中，代码常常写得不够好，比如把不必要的类之间的耦合以及和环境的耦合写进代码中。这样难以在不同环境下复用的代码，某些很小的错误可能会对系统中的其他类产生很大影响。为了使用模仿对象，必须从不同角度考虑代码，并且应用更合适的设计模式。

6.4 DbUnit 单元测试

6.4.1 DbUnit 简介

为依赖于其他外部系统(如数据库或其他接口)的代码编写单元测试是一件很困难的工作。在这种情况下，有效的单元测试必须隔离测试对象和外部依赖，以便管理测试对象的状态和行为。

开源的 DbUnit 项目，为以上的问题提供了一个相当好的解决方案。DbUnit 是对 JUnit 的扩展，专门针对数据库应用。使用 DbUnit，开发人员可以控制测试数据库的状态。进行一个 DAO 单元测试之前，DbUnit 为数据库准备好初始化数据；而在测试结束时，DbUnit 会把数据库状态恢复到测试前的状态。这是避免当一个测试用例修改了数据库，导致后续的测试失败等各种问题的合理途径。

DBUnit 具有 XML 文件和数据库数据相互转化的功能，可以导出数据库数据到 XML 文件中，也可以从 XML 文件中倒入测试数据集。DBUnit 还可以帮助验证数据库数据是否和期望的数据集相等。另外，它本身带了 REFRESH(如果不存在就插入，否则就更新，以主键为依据)，CLEAN INSERT(全部删除再插入，和 ReFresh 比，会把不在 XML 中的数据删除)等几种操作，又节省了编程的工夫。

DBUnit 中核心的类有 3 个：IDatabaseConnection、IDataSet 和 DatabaseOperation。这 3 个类也是 Dbunit 主要用来和用户交互的类。上述 3 个核心类的功能简单描述如下。

(1) IDatabaseConnection 接口。用于表示 DbUnit 与数据库连接。主要使用 Database-

Connection 类。DatabaseConnection 继承 AbstractDatabaseConnection 类，实现 Idatabase-Connection 接口，对与数据库的连接进行了简单的封装。

(2) IDataSet 接口。用于表示数据库中表的集合。主要使用以下 3 个类：

① FlatXmlDataSet 类：FlatXmlDataSet 读写纯 XML 文档。其中，XML 的属性对应一个表，属性名对应表名。表的元数据(metadata)由第一行导出。因此，第一行不可有 NULL 值，否则抛出 NoSuchColumnException 异常。

② XmlDataSet 类：与 FlatXmlDataSet 类相比，用法要烦琐冗长许多。

③ FilteredDataSet 类：用于对部分数据的验证。在某些情况下，测试人员只需要关注记录的部分内容，而不需要全部，使用 FilteredDataSet 类，可以根据需要检查表的特定部分。

(3) DatabaseOperation 抽象类。用于表示测试前后对数据库的操作。DataBase-Operation 封装了对数据库的操作，采用退化的工厂模式，直接使用子类。其中，两个最常用的操作是：REFRESH 和 CLEAN_INSERT。REFRESH 以迭代方式将数据集中的数据写入数据库。对于数据库中已经存在的行，根据数据集内容更新；对于不存在的行，根据数据集内容插入；对于已经存在但数据集中没有对应项的行，则不受影响。这个方法用于测试人员已知某些数据存在于数据库中的情况。CLEAN_INSERT 将首先删除表中所有行，再根据数据集插入对应行。这是确保数据库处于已知状态的最安全方法，用于确保数据库中只存在数据集中的数据。

使用 DBUnit 的好处主要体现在：

(1) 可以为面向数据库的应用准备一个相对稳定的外部环境，从而减少外部因素对代码运行结果产生的影响。

(2) 提供了一组方便的 assertion 方法，能简单地比较数据库的当前状态是否和预期情况相同。

详细内容请参考 http://dbunit.sourceforge.net。

6.4.2　使用 DbUnit

继续使用上一节描述的 ATM 例子。上一节使用了 MockDataConnection 类测试 ATMService 类。别忘了还有真正实现业务逻辑的 DBDataConnection 类，该类需要和数据库打交道，对数据进行操作。对该类的单元测试就要用到 DBUnit 了。

1. 准备测试数据

首先要为单元测试准备数据。使用 DbUnit，可以用 XML 文件来准备测试数据集，称为目标数据库的 Seed File，代表目标数据库的表名和数据。DBDataConnection 类是对 Account 信息进行操作，对应数据库中应该有一个 account 表。

清单 6-25 给出了例子中使用的测试数据，保存为 account_seed.xml 文件。元素名

account 对应数据库的表名，属性 cardNo、pwd 和 balance 对应表 account 的列名，属性值代表数据库中的具体数据。

清单 6-25：account 表测试数据 account_seed.xml

```
//account_seed.xml
<?xml version='1.0' encoding='UTF-8'?>

<dataset>
  <account cardNo="1" pwd="111111" balance="300"/>
  <account cardNo="2" pwd="222222" balance="500"/>
  <account cardNo="3" pwd="333333" balance="800"/>
</dataset>
```

在测试数据库之前，用这个 XML 文件中的数据去更新数据库，保持数据库中的内容完整。关于更新数据库的方法，在编写测试代码的实例中会进行讲解。

Seed File 可以手工编写，也可以用 DBUnit 提供的工具导出现有的数据库数据并生成。清单 6-26 给出的例程用于导出数据库中现有数据到 XML 文件。

清单 6-26：导出数据库数据例程

```
Class driverClass=Class.forName("com.mysql.jdbc.Driver");
Connection jdbcConnection=DriverManager.getConnection(
                    "jdbc:mysql://localhost:3306/test","root","root");
//IDatabaseConnection 封装对某个数据库的连接
IDatabaseConnection connection=newDatabaseConnection(jdbcConnection);
//导出数据库中的所有表
// createDataSet()创建一个代表整个数据库的数据集
IDataSet fullDataSet=connection.createDataSet();
FlatXmlDataSet.write(fullDataSet,new FileOutputStream("account_seed.xml"));
```

这样就把数据库中的所有的数据导出到 account_seed.xml 文件中了。通过使用这些数据，就可以准备任何我们所希望完成的测试用例。

DbUnit 有助于执行 JDBC 查询并获取它们的值。使用 DbUnit JDBC 包装器而不是纯粹的 JDBC 有以下几个理由：

(1) 可以用 SQL 查询创建一个 Dataset，并使用 DbUnit 的 assertion(断言)方法(具体将在后面描述)。

(2) 可以用 SQL 查询创建一个 Dataset，并将它保存为一个 FlatXmlDataSet。可以在以后将它重新装载到数据库中。

(3) 可以容易地从任何行中获取列的内容，无须进行迭代。

2. 创建 DbUnit 测试类

完成了数据准备，就可以开始设计测试类了。好的 JUnit 实践鼓励开发人员扩展基类

TestCase 以提供特殊(Specialization)行为。DbUnit 提供了自己的特殊版本的 DatabaseTestCase,通过直接继承它,开始写我们自己的测试用例。

首先必须重写父类 DatabaseTestCase 的两个虚函数：getConnection()和 getDataSet()。getConnection()方法返回 DbUnit 的一个到数据库的连接。DbUnit 的 DatabaseConnection 构造函数可以带一个 schema 名作为参数。这样,就不必在所有表名前面加上 schema 名的前缀了。getDataSet()方法用位于类路径上的一个 XML 文件的内容创建 DbUnit 数据集。清单 6-27 给出了两者的简单实现。

清单 6-27：覆盖 getConnection()和 getDataSet()

```
public class TestDBConnection extends DatabaseTestCase {
    ...
    protected IDatabaseConnection getConnection() throws Exception {
      Class driver=Class.forName("com.mysql.jdbc.Driver");
      Connection jdbcConnection=DriverManager.getConnection(
                  "jdbc:mysql://localhost/test","root","root");
      return new DatabaseConnection(jdbcConnection);
    }

    protected IDataSet getDataSet() throws Exception {
         return new FlatXmlDataSet(new
              FileInputStream("src/test/account_seed.xml"));
    }
    ...
}
```

还可以设置执行测试开始和结束时对数据库进行的操作,见清单 6-28。常用的操作有 REFRESH 和 CLEAN_INSERT,其意义如前所述。还有多种其他操作类型,可参考 http://dbunit.sourceforge.net/components.html。

清单 6-28：覆盖 getSetUpOperation()和 getTearDownOperation()

```
public class TestDBConnection extends DatabaseTestCase {
    ...
    protected DatabaseOperation getSetUpOperation() throws Exception {
        return DatabaseOperation.CLEAN_INSERT;
    }
    protected DatabaseOperation getTearDownOperation() throws Exception {
        return DatabaseOperation.NONE;
    }
    ...
}
```

需要特别注意的是，根据设定的操作不同，DBUnit 会更改数据库中的数据，尤其使用 CLEAN 的时候，会删除所有已存在的数据。所以使用这种测试前，必须确保数据的更改和删除不会带来其他影响。在多人开发的情况下，也需要确保没有其他人在同时使用数据库中数据。

下面就可以针对被测类的方法写相应的测试用例了。对于 DBDataConnection 中实现的 3 个方法 getAccount、addAccount 和 updateAccount 分别有一个测试用例，另外为了展示 DBUnit 中加入的对数据集(或表)进行比较的断言操作，用一个 testMe()方法说明，该方法并无实际意义，如清单 6-29 所示。

清单 6-29：设计测试用例

```
public class TestDBConnection extends DatabaseTestCase {
    DBConnection dbc;
    protected void setUp() throws Exception {
        super.setUp();
        dbc = new DBConnection();
    }
    ...
    public void testMe() throws Exception{
        IDataSet databaseDataSet = getConnection().createDataSet();
        //得到数据库中 account 表的数据集
        ITable actualTable = databaseDataSet.getTable("account");
        // 从 XML 文件中导入希望的数据集
        IDataSet expectedDataSet = new FlatXmlDataSet(new File("src/test/account_seed.xml"));
        ITable expectedTable = expectedDataSet.getTable("account");
        //比较两个数据集是否相等
        Assertion.assertEquals(expectedTable,actualTable);
    }
    public void testGetAccount() {
        Account a = dbc.getAccount("1");
        assertEquals("card No.","1",a.getCardNo());
        assertEquals("password","111111",a.getPassword());
        assertEquals("balance",300,a.getBalance());
    }
    public void testUpdateAccount() {
        Account na = new Account("1","111111",500);
        dbc.updateAccount(na);
        Account a = dbc.getAccount("1");
        assertEquals("balance",500,a.getBalance());
    }

    public void testAddAccount() throws Exception{
        Account a = new Account("4","444444",934);
```

```
            dbc.addAccount(a);
            ITable actualData=getConnection().createQueryTable("AD","select * from account where
    cardNo=4");
            assertEquals(1,actualData.getRowCount());
            assertEquals("card No.","4",actualData.getValue(0,"cardNo"));
            assertEquals("password","444444",actualData.getValue(0,"pwd"));
            assertEquals("balance",934,actualData.getValue(0,"balance"));
        }
    }
```

在setUp()中先初始化DBConnection对象dbc,在后面的每个测试函数中都要使用。

testMe()先导出数据库中原有的account表(这里的数据和account_seed.xml文件中的一样),再从account_seed.xml文件中创建新的数据集,然后用Assertion.assertEquals方法比较两个数据集是否相等。相应的函数原型如下:

```
public static void assertEquals(ITable expected,ITable actual);
public static void assertEquals(IDataSet expected,IDataSet actual);
```

这两个assert函数是DBUnit中新加入的。

testUpdateAccount()中调用dbc.updateAccount后,又使用dbc.getAccount得到结果数据;testAddAccount()中调用dbc.addAccount后,通过DBUnit提供的数据库连接查询数据库,得到结果数据集,再通过结果数据集可以很容易地获得各列的值(getValue),进而进行测试。

6.4.3 小结

数据状态维护非常必要,测试都是基于一定的数据状态进行的,所以保持状态,不干扰其他测试类的数据非常必要。数据状态被维护好,自动化测试就成为可能,尤其在涉及数据库测试时,如果需要人为去干涉和调整数据,那么肯定会造成测试效率的降低,持续集成测试等将无法进行下去,规划一下各个测试用例的数据状态,会为后来的测试带来非常好的便利。

参考资料

1. Control your test-environment with DbUnit and Anthill http://www-128.ibm.com/developerworks/library/j-dbunit.html
2. Effective Unit Testing with DbUnit
3. http://dbunit.sourceforge.net/

6.5 JUnit4 简介

JUnit4 对以前的 JUnit 框架进行了很大改进，其主要目标便是利用 Java5 的 Annotation 特性简化测试用例的编写。

Annotation 这个单词一般是翻译成元数据。元数据是什么？元数据就是描述数据的数据。也就是说，Annotation 在 Java 里面可以用来和 public、static 等关键字一样来修饰类名、方法名、变量名。修饰的作用描述这个数据是做什么用的，差不多和 public 描述这个数据是公有的一样。想具体了解可以看 Core Java2[6]。

6.5.1 一个小例子

先看一下在 JUnit 3.8.x 中我们是怎样写一个单元测试的。比如下面的类：

```
public class AddOperation {
    public int add(int x,int y){
        return x+y;
    }
}
```

要测试 add 这个方法，用 JUnit 3.8.x 写单元测试如下：

```
import junit.framework.*;

public class AddOperationTest extends TestCase{
      public void setUp() throws Exception {
      }
      public void tearDown() throws Exception {
      }
      public void testAdd() {
          System.out.println("add");
          int x=0;
          int y=0;
          AddOperation instance=new AddOperation();
          int expResult=0;
          int result=instance.add(x,y);
          assertEquals(expResult,result);
      }
}
```

可以看到上面那个单元测试有一些强制的规定，表现在：

(1) 单元测试类必须继承自 TestCase。

(2) 要测试的方法必须以 test 开头。

如果用 JUnit 4 来写上面那个单元测试，就不会这么复杂。代码如下：

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;
/**
 *
 * @author Mr. Bean
 */
public class AddOperationTest {

        public AddOperationTest() {
        }
        @Before
        public void setUp() throws Exception {
        }
        @After
        public void tearDown() throws Exception {
        }
        @Test
        public void add() {
              System.out.println("add");
              int x=0;
              int y=0;
              AddOperation instance=new AddOperation();
              int expResult=0;
              int result=instance.add(x,y);
              assertEquals(expResult,result);
        }

}
```

因为在 JUnit 4 中一个测试类并不继承自 TestCase(在 JUnit 3.8 中，这个类中定义了 assertEquals()方法)，所以必须使用前缀语法(举例来说，Assert.assertEquals())或者(由于 JDK5.0)静态地导入 Assert 类。这样一来，就可以完全像以前一样使用 assertEquals 方法(举例来说，assertEquals())。

可以看到，采用 Annotation 的 JUnit 已经不会要求必须继承自 TestCase 了，而且测试方法也不必以 test 开头了，只要以@Test 元数据来描述即可。

6.5.2 JUnit 4 的注解

从上面的例子可以看到在 JUnit 4 中还引入了一些其他的元数据，即注解，下面一一简单地介绍：

1. @Before

使用了该元数据的方法在每个测试方法执行之前都要执行一次。

2. @After

使用了该元数据的方法在每个测试方法执行之后要执行一次。

注意：@Before 和@After 标示的方法只能各有一个。这个相当于取代了 JUnit4 以前版本中的 setUp 和 tearDown 方法，当然你还可以继续叫这个名字，不过 JUnit4 不会要求一定这么做。

3. @Test(expected= * . class)

@Test 注解支持可选参数。它声明一个测试方法应该抛出一个异常。如果它不抛出或者如果它抛出一个与事先声明的不同的异常，那么该测试失败。例如，一个整数被零除应该引发一个 ArithmeticException 异常。

4. @Test(timeout=xxx)

该元数据传入了一个时间(毫秒)给测试方法，如果测试方法在制定的时间之内没有运行完，则测试也失败。

5. @ignore

该元数据标记的测试方法在测试中会被忽略。当测试的方法还没有实现，或者测试的方法已经过时，或者在某种条件下才能测试该方法(比如需要一个数据库联接，而在本地测试的时候，数据库并没有连接)，那么使用该标签来标示这个方法。同时，你可以为该标签传递一个 String 的参数，来表明为什么会忽略这个测试方法。比如：@Ignore("该方法还没有实现")，在执行的时候，仅会报告该方法没有实现，而不会运行测试方法。

6.5.3 小结

有很长一段时间，JUnit 简直成了事实上的单元测试框架标准。这个新版本提供了许多新的 API，而且现在还使用注解，所以使开发测试用例更为容易。事实上，该 JUnit 开发

者已经开始考虑新的注解问题。例如，可以在一个依赖于前提（举例来说，你需要在线执行这个测试）的测试用例上添加一个@Prerequisite 注解；或者添加一个能够指定重复次数及时限（举例来说，重复测试 5 次以确保真正出现了一个时限问题）的@Repeat 注解；或者甚至在@Ignore 注解上添加一个平台参数（举例来说，@Ignore(platform=macos)，这将只有在一个 Mac OS 平台上运行时才忽略一个测试）。

6.6 JUnit、Mock Object 和 DbUnit 的作业

（1）按照如图 6-18 所示的 ATM 业务模型，实现各个类（除了 DBDataConnection）。

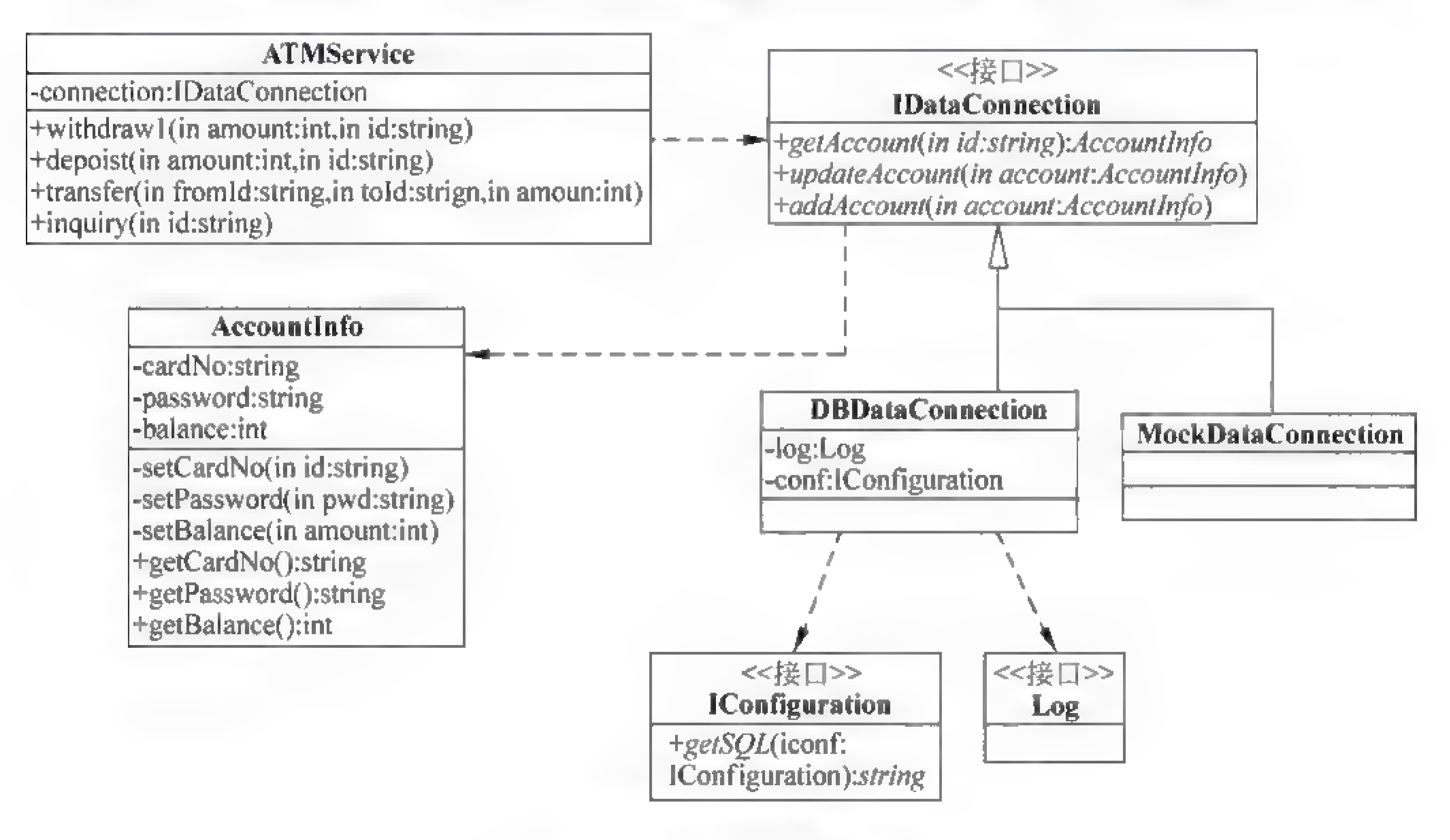


图 6-18 ATM 业务模型

每个类和接口的作用如下：

① AccountInfo 类保存账户信息，主要有 3 个数据域：cardNo（账号）、password（密码）和 balance（余额）。

② IDataConnection 数据访问接口，用于添加账户（addAccount）、获取账户（getAccount）和更新账户（updateAccount）。该接口有两个实现类：

- DBDataConnection 类用数据库实现，其中两个数据成员：log 为 Log 接口类型，conf 为 IConfiguration 接口类型。
- MockDataConnection 类为 Mock 类，用于测试 ATMService 类。

③ IConfiguration 接口用于读取配置的 SQL 语句（getSQL），也就是事先把 DBData

Connection 中要用到的 SQL 语句写到配置文件中，在程序中从配置文件中读取使用。

④ ATMService 类模拟实现 ATM 机的服务功能，包括取款(withdraw)、存款(deposit)、转账(transfer)、查询(inquiry)。需要使用 IDataConnection 接口提供的功能。

(2) 实现 ATMService 类后，在还没实现 DBDataConnection 类的情况下，实现 DBDataConnection 类的 Mock 类，用 Mock 对象完成对 ATMService 的单元测试。

① 手工实现 MockDataConnection 类。

② 使用 EasyMock2.2 版本提供的 Mock 机制。可参考下载包中的 Documentation.html 文档。

③ 使用 JMock 1.2 和 2.0 版本提供的 Mock 机制。用法参考 http://www.jmock.org/cookbook.html。

(3) 实现 DBDataConnection 类，使用 DBUnit 工具完成对 DBDataConnection 类的单元测试。Log 接口可使用 common-loggings 工具提供的实现。自己简单实现一个 IConfiguration 接口。

6.7 参考文献

[1] Ken Beck. 测试驱动开发. 影印版. 北京：中国电力出版社，2003

[2] Massol，V. 著. JUnit in Action(中文版). 鲍志云译. 北京：电子工业出版社，2005

[3] Rainsberger J. B.，Stirling. S. 著. JUnit Recipes(中文版). 陈浩，王耀伟，李笑译. 北京：电子工业出版社，2006

[4] JUnit A Cook's Tour，http://junit.sourceforge.net/doc/cookstour/cookstour.htm

[5] Erich Gamma，Richard Helm，Ralph Johnson. Design Patterns：Elements of Reusable Object-Oriented Software. *Addison-Wesley Professional*. November 10，1994

[6] Cay S. Horstmann，Gary Cornell. *Core Java* 2(*Seventh Edition*). Prentice Hall PTR，2004

第7章

回归测试

在软件开发、维护、升级的不断演进过程中，由于各种各样的原因，例如功能性/非功能性需求的变更、技术更新和软/硬件平台升级，使得软件系统经常发生改变。这些改变会给软件系统带来风险，因为改变传播效应(Change Propagation)可能会引入新的错误，有时甚至是致命的错误。在软件生命周期中的任何一个阶段，只要软件发生了改变就可能给该软件带来问题，而回归测试是一种验证已变更的系统的完整性与正确性的测试技术。因此，每当软件发生变化时，就必须重新测试现有的功能，以便确定修改是否达到了预期的目的，检查修改是否损害了原有的正常功能。同时，还需要补充新的测试用例来测试新的或被修改的功能。为了验证修改的正确性及其影响就需要进行回归测试。

回归测试(Regression Testing)是对之前已测试过并修改的程序进行的重新测试(Re-testing)，以保证该修改没有引入新的错误或者发现由于更改而引起的之前未发现的错误。回归测试通常在对被测系统(System Under Test，SUT)的第二个版本或后继版本进行测试时使用。通常回归测试的设计具有重复性。回归测试能应用于所有类型的系统，包括面向对象应用、电子商务或基于Web的应用系统。回归测试也被称为验证测试(Verification Testing)。

快速阅览：

什么是回归测试？ 回归测试是对之前已测试过、经过修改的程序进行的重新测试，以保证该修改没有引入新的错误或者由于更改而发现之前未发现的错误。

由谁来负责回归测试？ 软件开发人员、软件测试人员、软件维护人员都要参与回归测试。

为什么回归测试如此重要？ 每当软件发生变化时，就必须重新测试现有的功能，以便确定修改是否达到了预期的目的，检查修改是否损害了原有的正常功能。同时，还需要补充新的测试用例来测试新的或被修改了的功能。

回归测试步骤是什么？ 回归测试过程主要有7个步骤：提出修改需求、修改软件工件、

选择测试用例、执行测试、识别失败结果、确认错误和排除错误。

有哪些工件形成？ 在一些情况下，会生成回归测试计划、波及效应分析文档、重新确认测试用例、新生成的测试用例。在每一种情况下，要将回归测试结果存档以便将来软件维护时使用。

如何确保我们准确地完成了任务？ 尽管永远不能保证你已经执行了所要求的每一个回归测试，但能肯定测试已经发现了错误（并且已修正了这些错误）。另外，如果已经制定了一个回归测试计划，则可以检查来保证所有计划测试已被完成。

7.1 回归测试的特点

为什么需要进行回归测试呢？主要有以下4个方面的原因：

(1) 为了保证软件维护时，那些未更改的代码功能不会受影响。

(2) 建立各个功能区域、信息系统持续的维护与回归测试改进之间协作关系，使回归测试成为一个每月的常规活动。

(3) 为支持E2E测试（端到端测试）实施整个生命周期的测试。

(4) 建立一个测试框架，使各信息系统和各功能区域能够监控和维持用于回归测试的人力、时间等资源投入。

回归测试可以用在何处呢？在软件测试的3大层次都能使用回归测试技术：单元测试、集成测试、系统测试。这些测试用于报告3种不同类型的失败，而回归测试是一种在3个测试层次上都能使用的测试类型。

回归测试与一般测试有什么不同？下面从6个方面进行比较：测试计划的可获性、测试范围、时间分配、开发信息、完成时间和执行频率。

(1) 测试计划的可获性：一般测试都是先拿到系统规格说明书（Specification）、系统规格说明书的一个实现和带有一些测试用例的测试计划。从一定意义上说，这些测试用例都是新的，因为它们在之前从未用来执行程序。但是当要进行回归测试时，我们面临的可能是更改了的规格说明书、修改过的程序和一个需要更新的旧的测试计划。

(2) 测试范围：一般测试的目标是要检测整个程序的正确性，而回归测试目标是要检测被修改的相关部分正确性以及它与系统原有功能的整合。

(3) 时间分配：一般测试所需时间通常是在一个产品开发之前都被预算好的，但是回归测试所需的时间（尤其是修正性的回归测试）是不包含在整个产品进度表中的。

(4) 开发信息：在一般的测试中，关于开发的知识、信息都随时可以获得。而回归测试可能会在不同的地点和时间进行，所以需要保留开发信息以保证回归测试的正确进行。

(5) 完成时间：回归测试完成所需时间通常比一般测试所需时间少，因为回归测试只

需测试程序的一部分。

(6) 执行频率：一般测试是发生频率很高的一个活动。回归测试在一个系统的生命周期内往往要多次进行，一旦系统经过修改就需要进行回归测试。

7.2 回归测试的过程

回归测试过程主要有7个步骤：提出修改需求、修改软件工件、选择测试用例、执行测试、识别失败结果、确认错误和排除错误。

(1) 提出修改需求：软件可能因为要改正一个错误(Bug)而被修改，或者根据需求规格说明书或者设计说明书而被修改。

(2) 修改软件工件(Software Artifact)：为了满足新的需求或者改正错误而对软件工件进行修改。

(3) 选择测试用例：通过选择和有效性重确认过程获取正确的测试用例集，而不是要使测试用例数目最小化。有时一些测试人员会不进行有效性重确认而直接使用所有的已有测试用例。

(4) 执行测试：这一步骤通常是自动化运行的，因为会有大量的测试用例要执行。测试执行历史(遍历过的路径和被调用的过程、操作)都要被记录下来，这样能给将来的测试做参考。

(5) 识别失败结果：如果测试结果与预期结果不一致，则有必要检查是测试用例错误，还是代码错误，或者两者都有错误。如果测试用例在早期没有进行有效性重确认，那么这时就该进行有效性重确认工作，尤其是那些已经导致失败的用例。下面将进一步讨论有关重新确认测试用例的有效性过程。

(6) 识别错误：精确定位是哪个版本中的哪个组件以及哪些修改导致的失败。在检查测试结果以识别失败时，如果使用的测试用例的有效性在执行之前已被确认过，那么任何与预期结果的明显偏离都表明了软件存在一个潜在错误。如果所使用的测试用例的有效性在之前未被确认，那么任何测试用例失败可能意味着要么是测试用例的不正确，要么是程序的错误，要么两者皆有。

(7) 排除错误：一旦识别了导致失败的组件，程序员就必须对这一组件进行排除错误工作。当一个错误被检测到后，可以采取以下几种行动来改正错误：

① 改正错误后，提交一个新的程序修改卡(PMC)。

② 移去引起错误的修改卡(PMC)，即修改错误。

③ 忽略错误。

下面就重新确认测试用例和识别错误做进一步讨论。

7.2.1 重新确认测试用例

测试用例的有效性重确认过程主要靠手工操作进行，目标是为了识别出对于更改了的软件现有的已经不再有效的测试用例。在有效性重确认过程中，测试的输入和它的预期结果都应该被检查。如果测试用例的输入不再有效，那么这一测试用例就要被丢弃或作为一个消极测试用例(Negative Test Case)；如果它的输入仍然有效，但是预期结果不再有效，那么就需要产生新的、相应的预期结果。

重确认测试用例需要人工检查需求规格说明书、测试策略和已存在的测试用例，所以可能很费时。对于黑盒测试，如果在功能性需求和测试用例间维护一种可回溯性(Traceability)，那么有效性重确认的效率会高得多；对于白盒测试，因为对软件的修改可能会导致设计和编码上的修改，所以需要生成新的测试用例，或者更改已存在的测试用例以达到一定的覆盖率标准。

7.2.2 识别错误

在识别错误时，为了确认软件中失败的组件，在列表中所有列出来的模块都有可能是要查找的目标。但这不是一个万无一失(Failure-Safe)的过程，虽然它有可能帮助发现有错误的部件。下面介绍一种较为系统性的识别错误方法——组测试(Group Testing)。

一组模块可能单独地运行时都能正常、正确地工作，但是当它们与软件中的其他组件集成到一起时，这个组合体就可能使在单个组件测试能通过的测试用例失败。可以使用组测试(Group Testing)进行错误识别，在寻找有错误的部件时，可以利用可获得的软件的不同版本来帮助发现错误的部分[1]。

如何进行组测试呢？假设一个程序有5个组件，分别是A、B、C、D和E，在早期的测试中，各个组件都有一个可以信赖的版本，即正常工作版本。假设现在A、B、C被修改了，而D、E没有被修改，仍然是之前能正常工作的组件。假设这一新版本软件未能通过几个测试用例，我们需要发现错误的组件所在。

首先使用新版本的A组件和原来能正确工作可以信赖的B、C、D、E组件来重新运行那些失败的测试用例，来观测这样配置是否导致测试用例失败。如失败，则有很大可能性是新版本的A组件导致的。若不是，则再使用新版本的B组件和原来能正确工作的A、C、D、E组件来重新运行那些失败的测试用例，以确定是否是B组件导致的失败。如不是，这个过程还要继续进行下去，直到所有新版本的组件全部实验过，如图7-1所示。

用这个方法，可以决定是否是新版本的A、B、C中的某一组件有错。也可以重复多次运行测试用例以查明导致用例失败的具体组件。问题是这个方法安全吗？回答是“不安全”。但是它可以进行自动化测试，也已经被证实这个方法很有用。

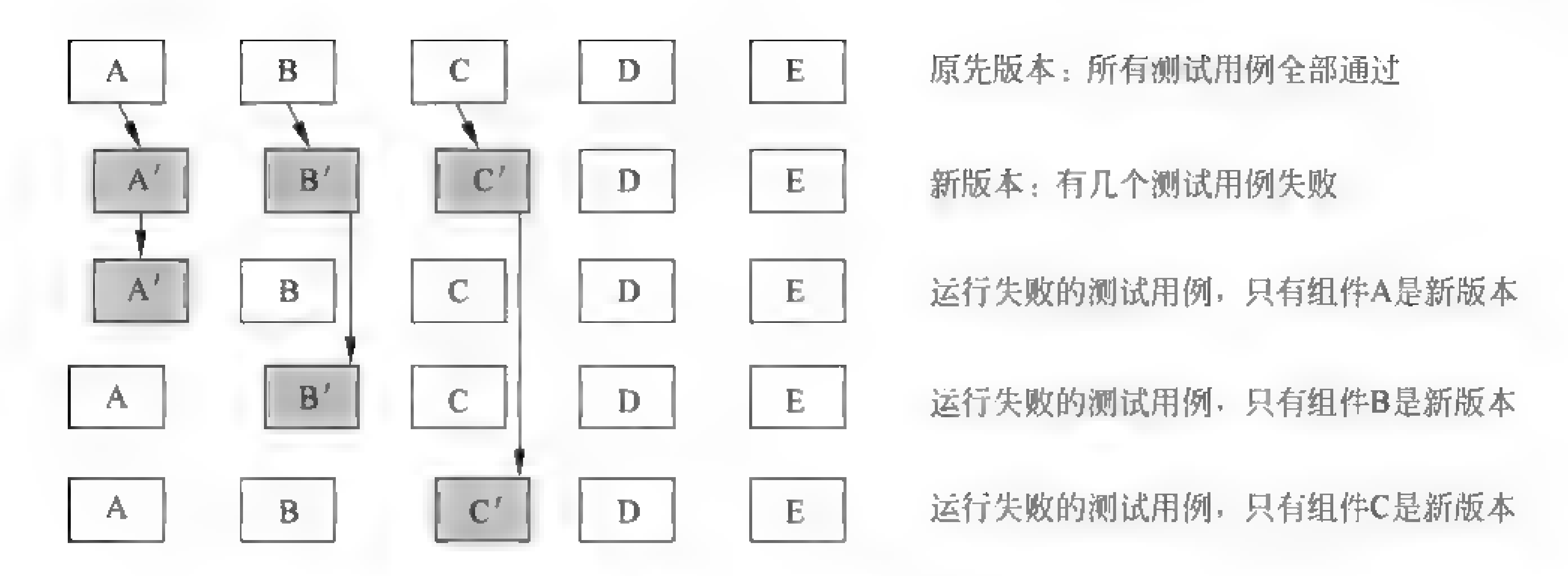


图 7-1 组测试示例

7.3 回归测试的策略

回归测试的策略有两个主要方法，分别是：

(1) 全部重新测试，即将之前所有的测试用例全部重新执行。

(2) 有选择地重新测试，即选择和使用已存在的测试用例的一个子测试用例集。

也有资料称这两种方法为两种"模式"[3]。在一些情况下，选择被测系统的所有测试用例进行重新测试是不可行的，而在另一些情况下，可以使用选择性重新测试。

- 全部重新测试方法：优点是不需要花精力去选择要执行的测试用例，所以当测试用例的数目不太多或者一个系统大部分被改变时，使用该方法是合适的；缺点是当测试用例的数目很多，而对于一个系统改动是很微小的，那么全部重新测试就很浪费。
- 有选择的重新测试方法：优点是当测试用例的数目很多，而且系统的更改只是一小部分，那么这个方法是很有用的；缺点是需要费精力对测试用例进行选择。当全部测试用例的数目不是很大，或者对系统的更改也很多，那么这个方法就不适合了。

有选择的重新测试方法中需要对测试用例进行选择，选择的最常用算法是选择所有与某个特定模块相关的所有测试用例，及所有集成测试用例。而对那些固定的特征，应保持它们不变。在识别相关的测试用例时，依据可追溯性原理，从需求到代码，然后再到测试用例，都要可追溯。所有黑盒测试和白盒测试的测试用例都要做到可追溯。由于软件工件间的依赖性，所以需要使用波及效应分析来确认那些被影响到的部分。下一节介绍波及效应分析(Ripple Effect Analysis，REA)。

7.4 波及效应分析

在介绍什么是波及效应分析之前，首先说明几个事实和相关问题。主要的事实有：

(1) 软件是会被修改的。

(2) 与软件相关的所有东西(需求款项、设计款项、代码、测试用例、文档)都可能被修改。

(3) 修改在任何时候都可能发生，比如在软件开发阶段和软件维护阶段。

修改中遇到的问题是：

(1) 需要定位改变部位。

(2) 改变的完全性问题。

(3) 改变有效性重确认问题。

(4) 完整性问题。

(5) 追踪问题。

波及效应分析是为了发现所有受影响部分和发现潜在的受影响部分，以保证软件发生改变后仍然保持一致性与完整性。因为软件中所有类型的工件都可能变化，需要对所有工件进行波及效应分析。波及效应分析共有4种类型：

(1) 需求的波及效应分析。

(2) 设计的波及效应分析。

(3) 代码的波及效应分析。

(4) 测试用例的波及效应分析。

对于各阶段之间同样也需要进行波及效应分析，也就是从需求文档到设计文档，再到代码，最后到测试用例。

7.4.1 波及效应分析步骤

波及效应分析是一个迭代过程，直至不再有任何波及。具体步骤如下：

(1) 实施初始的改变。

(2) 识别潜在的受影响的组件。

(3) 决定这些受到潜在影响的组件中哪些需要改变。

(4) 决定如何进行这些改变，对于每一个改变都要从第(1)步开始重复。如果没有要进行改变的就结束。

这4个步骤可以用如图7-2所示的流程图表示。

波及效应分析在第(1)、第(3)和第(4)步时需要用户的参与，第(2)步的识别潜在波及可以自动化进行。对于代码，主要有两个技术来识别潜在波及：

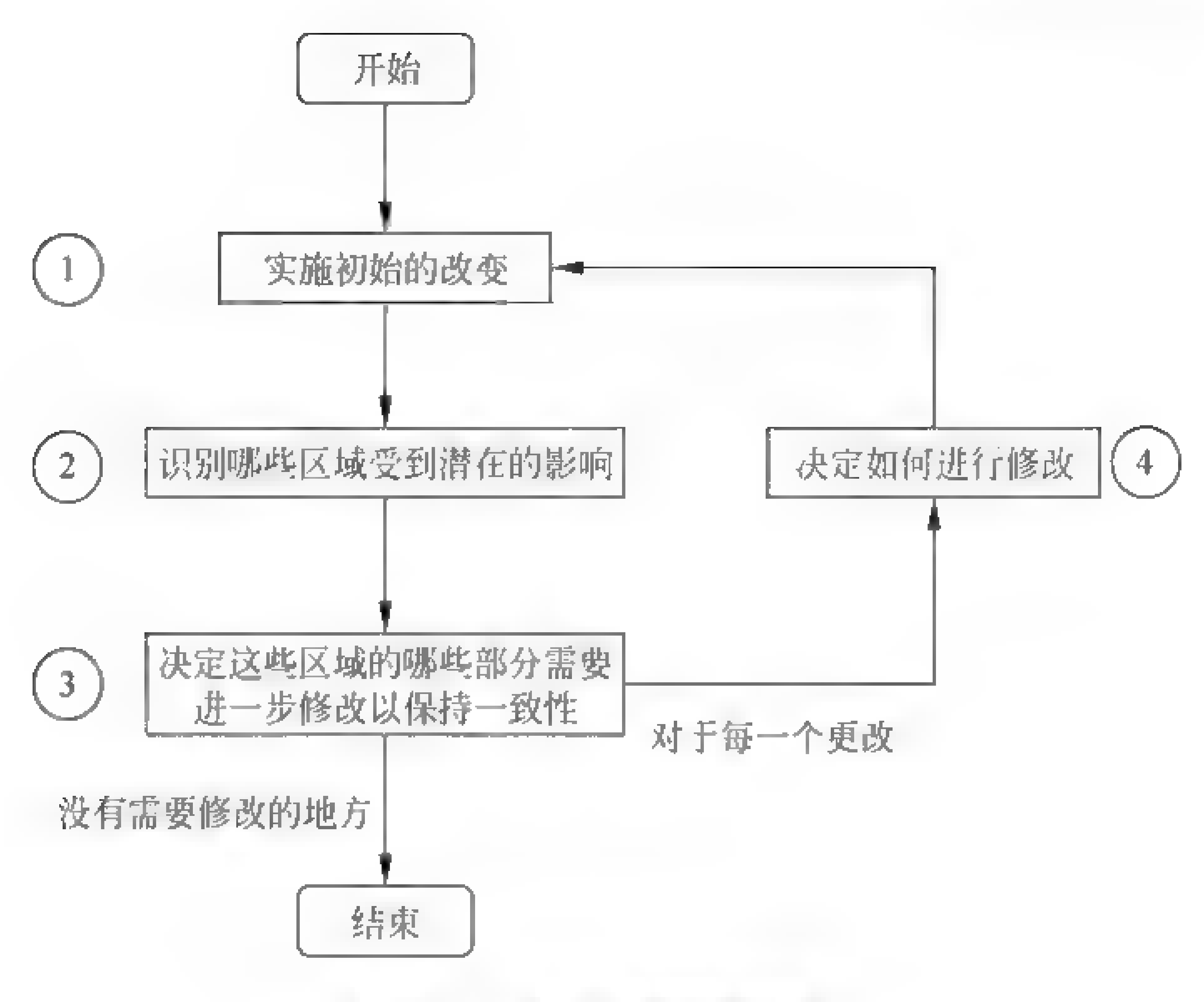


图 7-2 波及效应分析过程

(1) 字符串匹配或者交叉引用。

(2) 程序切片。与程序切片方法相比,字符串匹配不需要考虑程序执行 —— 它需要人类智慧来进行 REA 操作。绝大多数的再工程工具都是基于字符串匹配的。程序切片方法能确认直接和间接的波及,并可以自动产生巨大的程序切片。后面将介绍使用程序切片的技术来自动地进行潜在波及影响识别。

程序切片利用数据依赖和控制依赖来识别潜在的波及。

- 控制的依赖性。控制可由 if-then-else、while、for 和顺序型语句以及面向对象程序中的消息传递所引起。如果一个 if-then-else 语句的条件改变了,那么 then 部分和 else 部分都要因为条件改变而被影响。
- 数据的依赖性。数据的操作主要有3个:数据的使用、数据的定义和数据空间的释放,往往只考虑数据的定义和使用。在确认数据流前需要确认执行路径,然后查找每一个"定义-使用"对。在进行控制依赖性确认前是无法进行数据依赖性的确认的,可能会发现没必要的数据依赖性。

波及效应有直接波及(Direct Ripples)和诱发波及(Induced Ripples)。直接的波及是那些被初始的更改影响的——它们对于更改点有直接的数据或控制依赖。诱发波及是由直接波及和其他诱发波及引起的,如图7-3(a)所示,"初始改变"使得组件 1、2、…、n 等受到直接的波及,即这些组件对于更改有数据或控制依赖。而受到直接的波及组件又依赖于组件 11、$1k_1$、21、$2k_2$、$n1$、nk_n 等。对于这些组件的影响成为诱发波及。这种诱发波及可以一直持续下去。

被直接波及的组件是波及效应分析过程中要进行进一步更改的第一候选集。如果直接波及后不需要进一步地更改，如图7-3(b)所示：组件2已经确定不需要进一步的更改，那么诱发波及就不需要分析。这样可以节省波及效应分析时间。所以在实际使用过程中，建议使用这样的增量分析方式，而不是枚举方式。

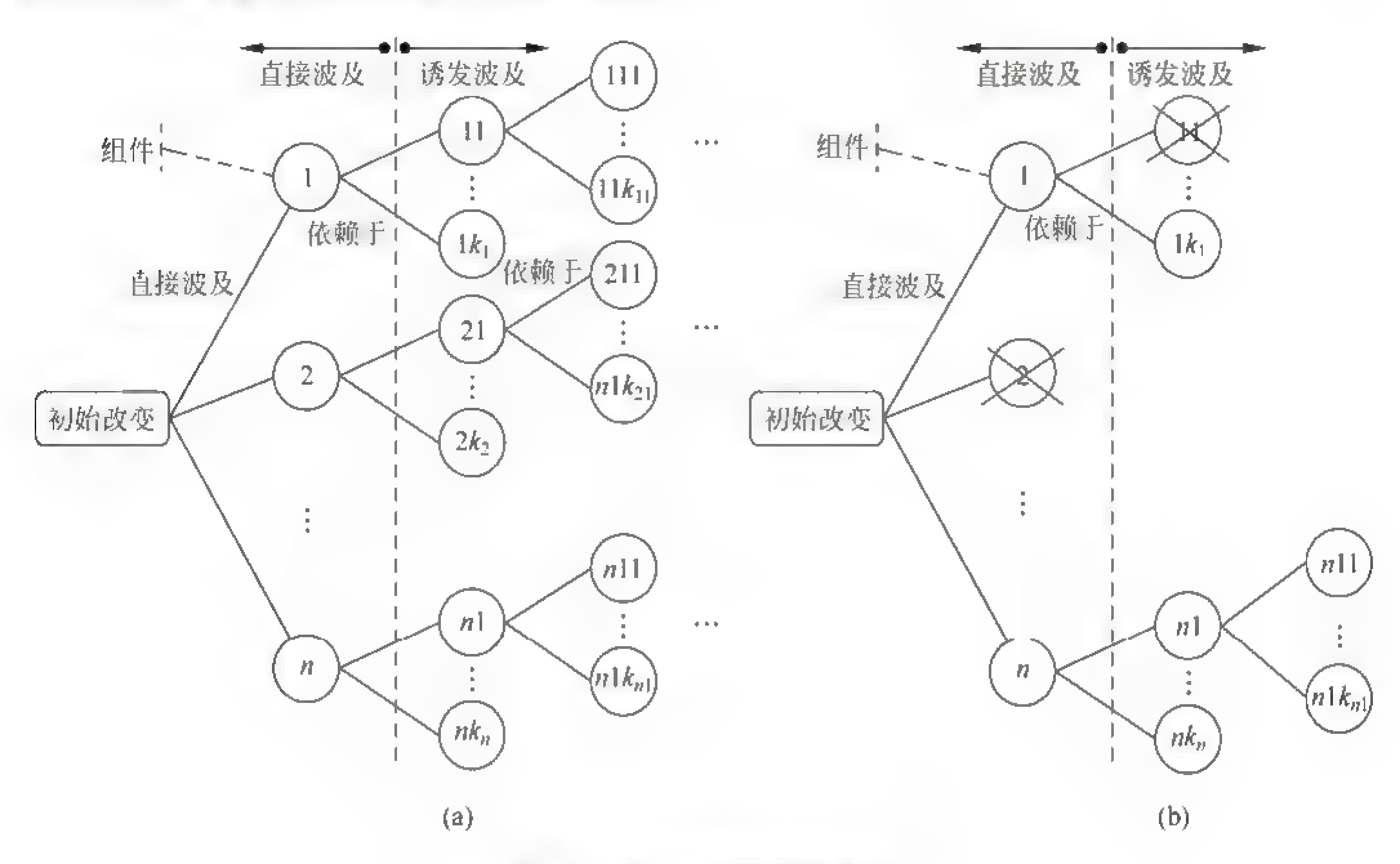


图7-3 直接波及和诱发波及

7.4.2 程序切片

上节讲到波及效应分析第(2)步的识别潜在波及可以利用程序切片技术自动化进行。本节介绍程序切片的技术如何自动地进行潜在波及影响识别。注意这里所说的"潜在"意义。自动化只能识别"可能的"，然而确认工作还是需要"人工"来完成。

程序切片(Program Slicing)定义切片标准(Slicing Criteria)来说明所关心的切片语句开始点和一些变量。程序切片结果产生一个语句集，这些语句影响到切片标准中被定义的语句的变量值。程序切片最早是由Weiser[2]提出的。

切片分向前和向后两个方向。向后的程序切片(Backward Program Slicing)是指给定一个语句号和一个变量集，它能产生所有语句，当程序执行到给定的语句时这些语句将影响给定语句中的变量集。向前的程序切片(Forward Program Slicing)是指给定一个语句号和一个变量集，它能产生所有语句，当程序恢复执行给定语句时，这些语句将受到给定语句中的变量集的影响。

如图7-4所示的一个程序切片的例子：图的左上角是源代码，有5条语句。图的右上角是数据流信息，表示各语句定义和使用的变量。图的左下角是向前的程序切片的标准及其结果。标准<1,{a}>表示从语句1开始，关注的变量是a；程序切片是从语句1顺着程序执行方向，切片的结果是语句集{1,3,4,5}。第1条语句里变量a被定义，其语句自然受影响；第3条和第4条语句直接使用变量a，要受其变化影响；第5条语句中的c和d使用变量a，从而间接地受到a的变化影响。程序切片的结果的语句集中没有第2条语句，这是因为第2条语句不直接或间接地使用变量a，因而不受其变化影响。

图的右下角是向后的程序切片的标准及其结果。标准<5,{d}>表示从语句5开始，关注的变量是d；程序切片是从语句5逆程序执行方向，切片的结果是语句集{1,2,4,5}。程序切片是从第5条语句开始的，第5条语句被收入结果集中；第4条语句定义了变量d，对于使用变量d的第5条语句有影响；而第1条和第2条语句中分别定义的变量a和变量b，被第4条语句用来定义变量d，从而影响第5条语句。

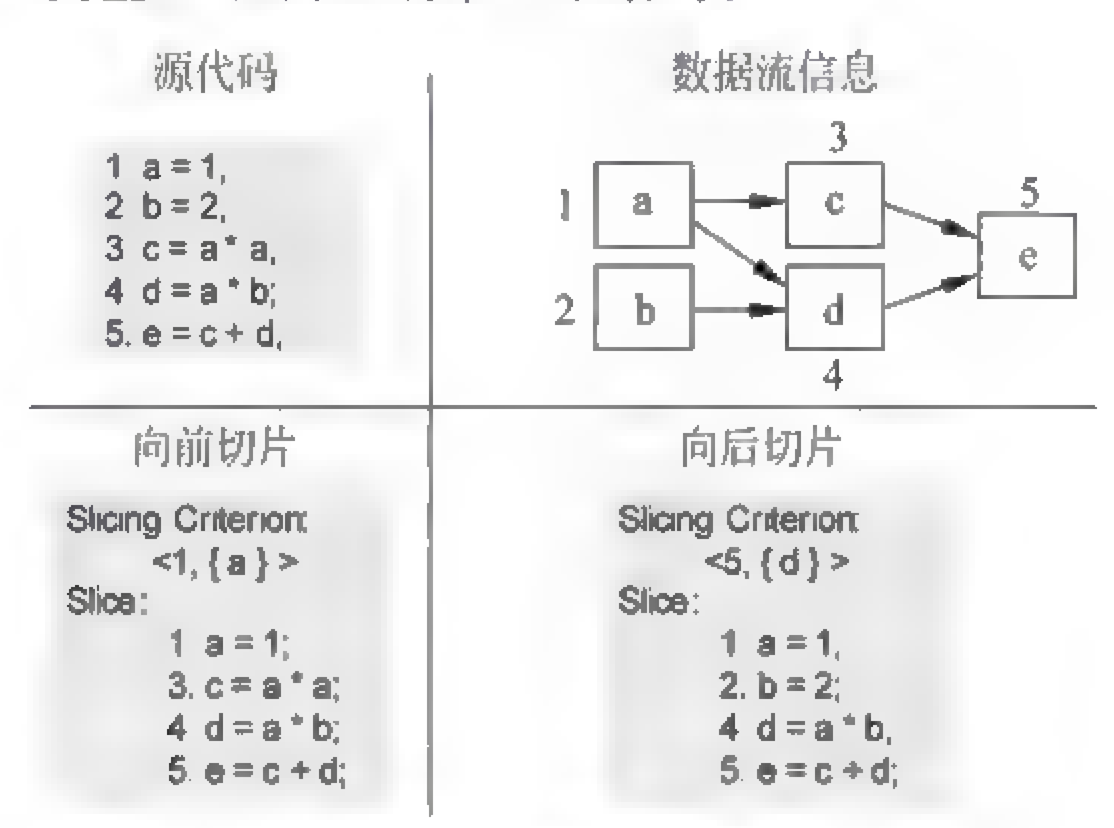


图7-4 程序切片示例

波及效应分析过程中可以应用程序切片技术。向前切片常常用于在波及效应分析过程中的第(2)步来识别潜在的波及；向后切片也被用于第(2)步中来识别各种类型的改变（如定义、使用和控制）；向后切片在第(4)步决定将如何影响其他的改变时很有用。

程序切片方法往往生成数量很大的切片集。通用化的程序切片可以将切片数量限制在一个比较合理的范围内。在波及效应分析中比较有用的两种限制方法是进行深度限制和边界限制。深度限制是将切片中的语句限制在离更改点一个特定距离内以达到限制切片数量。边界限制是将切片限制在一个特定的模块或函数内。

7.5 回归测试的花费

回归测试时间主要消耗在以下几个方面：

(1) 为了测试那些新改变的代码，要生成测试用例。

(2) 对原有的测试用例组进行有效性重确认。

(3) 执行测试用例组。

(4) 比较测试用例的执行结果与预期结果。

(5) 回溯失败,查明导致失败的模块或修改。

对于回归测试给出两点实用性建议。一是对于那些常常要进行回归测试的人员来说,使用工具是有好处的。

(1) 测试执行工具:因为有巨大数量的测试用例,所以使用自动测试执行工具是必需的。

(2) 测试结果比较器:在确认测试失败时它是很有用的。

(3) 配置工具:它能跟踪其中的数据、控制依赖性。

(4) 测试管理工具:跟踪测试现状。

二是在回归测试中应该考虑两种依赖性来帮助分析波及效应。

(1) 模块依赖:软件中的一个模块会依赖于其他模块,因为要使用其他模块来完成自己的任务。

(2) 修改依赖:它发生在软件的项目组人员要跟踪程序修改卡(PMC)时。

7.6 总结

执行回归测试是一种技术,它提供一种快速、方便的方法来决定代码的修改是否改变或破坏了现有的功能。回归测试与一般测试比较,有 6 个方面不同:测试计划的可获性、测试范围、时间分配、开发信息、完成时间和执行频率。

回归测试过程主要有 7 个步骤:提出修改需求、修改软件工件、选择测试用例、执行测试、识别失败结果、确认错误和排除错误。回归测试有两个主要方法:

(1) 全部重新测试,即将之前所有的测试用例全部重新执行。

(2) 有选择地重新测试,即选择和使用已存在的测试用例的一个子测试用例集。

对于被改变的代码,需要生成新的测试用例。波及效应是研究改变给软件所带来的直接波及和诱发波及,以保证软件发生改变后仍然保持一致性与完整性。波及效应分析是一个迭代过程。有的步骤可以自动化执行,而有的步骤必须手工进行。程序切片能自动地帮助进行识别潜在波及影响。

回归测试要消耗很多时间,建议那些常常要进行回归测试的人员,使用各种工具来提高效率,并利用模块依赖和修改依赖来帮助分析波及效应。

7.7 参考文献

[1] A. K. Onoma, W. T. Tsai, M. Poonawala. Regression Testing in an Industrial Environment. *Communications of the ACM*. Vol. 41, No. 5, 1998, pp. 81~86

[2] Mark Weiser. Program Slicing. *Proc. 5th Int. Conf. on Software Eng*. New York: IEEE, 1981, pp. 439~449

[3] R. V. Binder. *Testing Object oriented System: Models, Patterns, and Tools*. Addison Wesley, 1999

7.8 思考与练习

1. 试用自己的话，描述什么是回归测试。
2. 和一般测试相比，回归测试有什么特点？
3. 回归测试的步骤是什么？测试用例的有效性重确认可以实现自动化吗？
4. 用“组测试”方法描述如何识别组件缺陷。
5. 回归测试的两种模式是什么？是否还有其他模式(举例说明)？
6. 什么是波及效应分析？它的适用范围是什么？为什么说波及效应分析是一个迭代过程？
7. 什么是向后的程序切片？什么是向前的程序切片？作为切片标准中的语句一定会在切片结果集中吗？为什么？
8. 回归测试有哪些消耗？

7.9 进一步阅读

H. Agrawal, J. R. Horgan, E. W. Krauser Incremental Regression Testing. *Proceedings of the Conference on Software Maintenance*. 1993, pp. 348~357

G. Baradhi, N. Mansour. A Comparative Study of Five Regression Testing Algorithms. *Proceedings of Software Engineering Conference*. Australian, 1997, pp. 174~182

J. Hartmann, D. J. Robson. Techniques for Selective Revalidation. *IEEE Software*. Vol. 7, No. 1, 1990, pp. 31~36

李丹. 软件回归测试及其实践. http://www.testage.net/TestTech /FT/200601/184.htm

第8章

基于状态的软件测试技术

基于状态的软件测试技术是一种基于模型的测试技术(Model-Based Testing,MBT)。MBT 利用系统需求模型和特殊功能模型,自动生成有效的测试用例。一种定义明确的有限状态机(Finite State Machine)常用来帮助描述系统的行为。从测试的角度看,MBT 技术带来的最大好处是针对可用的有限状态机或其变种,生成测试用例只需遍历状态机。各种图论算法可以用来遍历这个模型(如 shortest path、N-states、all-states、all-transitions 等)。除此之外,MBT 技术还带来其他许多益处,如增强开发人员与测试人员之间的沟通、尽早暴露规范与设计中的不明确之处、自动生成很多无重复的有用的测试用例、减轻由于变化了的需求带来的更新测试集的工作、提高评估回归测试的能力等。

MBT 技术带来的所有益处都在一个假设条件下,即所建立的被测系统的状态机"正确地"描述了系统的行为;换句话说,模型的质量决定着 MBT 技术的成败。因此本章的前两节将分别介绍状态转换图模型和状态图模型。如果读者已经熟悉这两类状态机模型,可以直接从第三节开始阅读。

长期以来,状态转换图(State Transiation Diagram,STD)作为一种图形化标记用来描述计算机系统。20 世纪 50 年代中期,G. H. Mealy 和 E. F. Moore 同时引入了两种状态转换图的基本模型,这两种模型在硬件设计领域一直发挥着重要的作用。近些年来,这两种模型得到了广泛扩展,增加了对诸如层次结构(Hierarchy)、适时(Timing)和通信这些方面的表达能力,使得可以使用这两种模型对复杂的软件系统进行建模。例如,使用这类状态模型(如 Harel/UML 状态图)可以对实时系统、嵌入式系统、Web 交互性系统行为进行模拟、验证及测试。本章介绍基于状态模型的软件测试技术,在这类系统中得到广泛的应用。

快速阅览:

什么是基于状态的测试? 基于状态的软件测试是一种基于模型的测试技术。也就是通过建立描述系统行为的状态机,来自动生成测试用例。

由谁来负责基于状态的测试? 软件开发人员、软件测试人员都要参与基于状态的测试。

为什么基于状态的测试如此重要？ 基于状态模型的软件测试技术，广泛应用于实时系统及嵌入式系统的测试。

基于状态的测试步骤是什么？ 基于状态的测试过程主要有3个步骤：创建图形化规格说明书、产生中介规格说明书和生成测试规格说明书。

有哪些工件形成？ 在一些情况下，会生成基于状态的测试计划、被测系统状态机模型、生成测试用例的有向图、生成的测试用例。在每一种情况下，要将基于状态的测试结果存档以便将来软件维护时使用。

如何确保我们准确地完成了任务？ 尽管永远不能保证你已经执行了所要求的每一个回归测试，但能肯定测试已经发现了错误（并且已修正了这些错误）。另外，如果已经制定了一个基于状态的测试计划，则可以检查以保证所有计划测试已被完成。

8.1　状态转换图

图（Graph）和图形化标记（Graphic Notation）在软件规格（Software Specification）的陈述、分析和软件设计方面起着显著的作用：从数据流图到实体-关系图，从组合结构图（Modular Structure Diagram）到Petri网，图形的应用范围非常广。在不同的图形表示中，节点和箭头有不同的含义（Interpret）和注解（Annotate）。状态转换图是一个简单有向图，图中的节点代表系统的状态，箭头代表状态之间的转换。

Mealy和Moore奠定了有限自动机理论（Finite Automata Theory）的基础[1][2]。Moore的论文中给出了有限自动机实验的概念，得出了通过外部实验（External Experiments）可以得到有关有限自动机内部状态的结论。外部实验包括给自动机某些输入，观测其输出结果。Moore证明了大量关于有限自动机等价和简化（Reduction）方面的定理，这些定理已经成为自动化理论教科书上的标准内容。Mealy把Moore的概念应用到数字电路的合成（Synthesis）和简化（Reduction）上。Mealy有限自动机是Moore有限自动机的一个变体。因为Mealy有限自动机更符合我们的意图，所以先做介绍；然后再把Moore有限自动机作为其变体进行介绍。

定义8.1： 一个Mealy有限自动机是一个六元组$(S,I,O,\delta,\gamma,s_0)$，其中S为有限状态集，I为有限输入字符表，O为有限输出字符表，$\delta: S\times I\to S$为状态转换函数，$\gamma: S\times I\to O$为输出函数，$s_0\in S$为初始状态。

根据定义8.1，δ函数和γ函数分别描述了有限自动机接受一个输入后转换成的新状态和输出结果。

举一个简单的例子，考虑如图8-1所示的状态转换图。它描述了一个受限栈（Bounded Stack）的Mealy有限自动机模型。该受限栈最多只能从集合{a,b}中压入两个元素。图中有7个状态（节点），以栈中元素来标识（ε表示空栈）。短箭头表明ε为初始状态。输入字符

表为集合{$push_a$, $push_b$, pop, top}，其中 $push_a$ 和 $push_b$，分别表示向栈中压入 a 或 b，pop 表示栈顶元素出栈，top 表示返回栈顶元素。

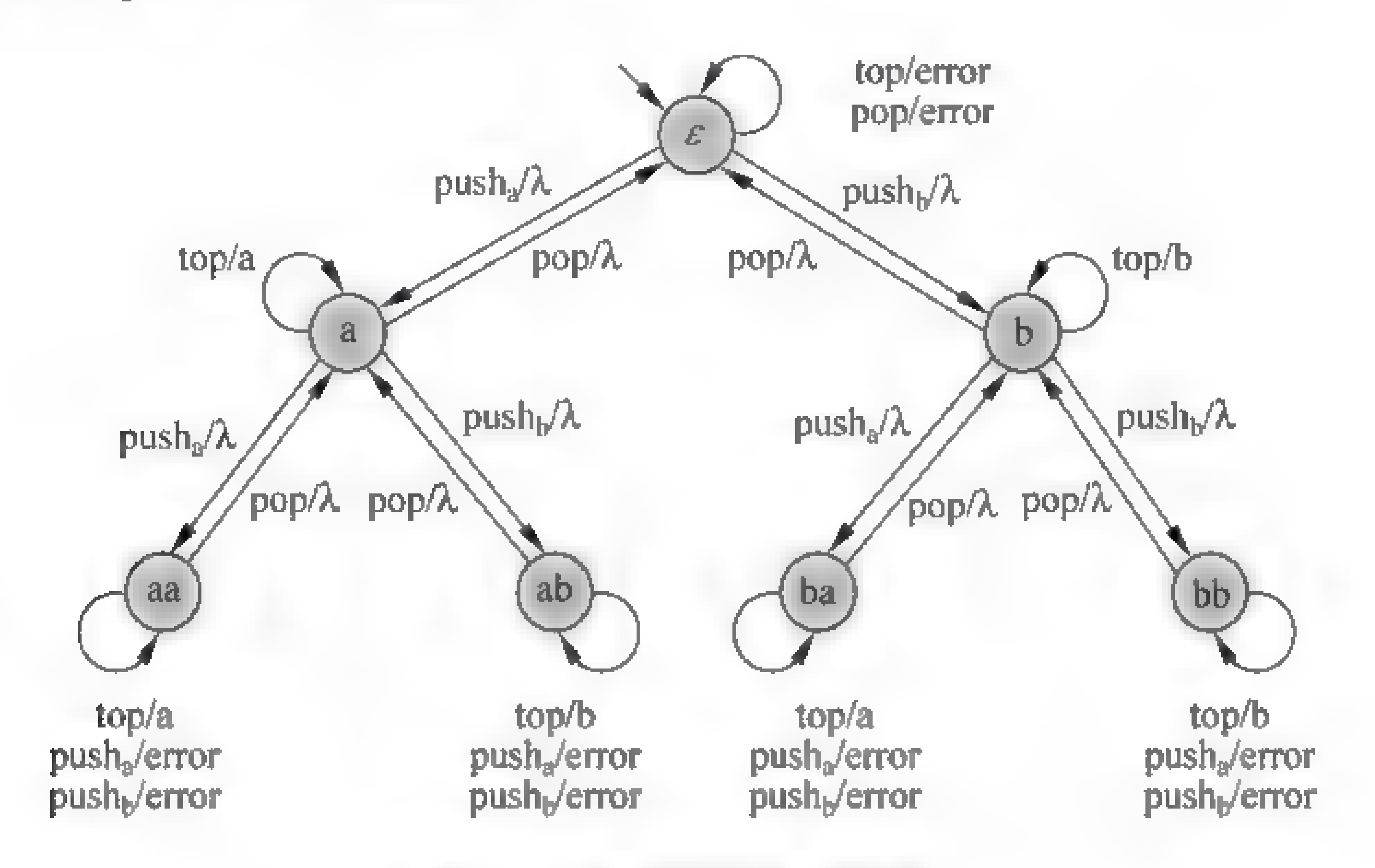


图 8-1 受限栈状态转换图

输出字符表为集合{a, b, λ, error}。δ 函数和 γ 函数可直接从图 8-1 中读出：标签(Label)x/y，其中 $x\in I$，$y\in O$，从状态 s 转换到状态 t 对应函数 $\delta(s,x)=t$ 和 $\gamma(s,x)=y$。因此，在状态 a 时输入 $push_b$ 导致转换到状态 ab 并且输出为 λ(λ 可以有多种解释，可以表示无关紧要的值(don't-care value)，无意义的消息(mute message)，或者表示没有输出(absence of output))。在一个空栈上应用 pop 和 top 操作会输出 error；类似地，在一个满栈上应用 $push_a$ 或 $push_b$ 操作也会输出 error。状态 a 上的环形标记 top/a 表示 top 操作可以反复应用到只包含 a 元素的栈上，其输出结果是 a。

当某人画出一个 Mealy 有限自动机的状态转换图时，他所定义的语义是什么呢？这就引出一个相关的问题：什么情况下两个 Mealy 有限自动机是等价的？一旦定义了 Mealy 有限自动机的语义，那么具有相同语义的两个有限自动机被认为(在语义上)是等价的。

在给出对语义的定义之前，先引入下面的符号：

- r 表示空序列(空串)。
- T^+ 表示集合 T 中元素的非空有限序列。
- T^* 表示集合 T 中元素的有限序列($T^*=T^+\cup\{r\}$)。

集合 T 中元素的串接或序列就是把元素依次排列起来组成的列(Juxtaposition)。

定义 8.2：一个 Mealy 有限自动机 $\Sigma=(S,I,O,\delta,\gamma,s_0)$ 的行为抽象(语义)是函数 g_Σ：$I^+\to O$，g_Σ 由下面的递归等式定义，其中 d_Σ：$I^*\to S$ 为辅助函数，$x\in I$，$t\in I^*$，$d_\Sigma(r)=s_0$，$d_\Sigma(tx)=\delta(d_\Sigma(t),x)$，$g_\Sigma(tx)=\gamma(d_\Sigma(t),x)$。

由定义 8.2 易知，两个 Mealy 有限自动机，$\Sigma=(S,I,O,\delta,\gamma,s_0)$ 和 $\Sigma'=(S',I,O,\delta',\gamma',s_0')$ 是等价的，当且仅当 $g_\Sigma(t)=g_{\Sigma'}(t)$，$t\in I^+$。

换句话说，$d_{\Sigma}(t)$是对有限状态机输入串 t 后转换成的状态，而 $g_{\Sigma}(t)$是有限状态机的最终输出(字符)。如果两个有限自动机对相同的输入串产生相同的最终输出(字符)，则说明这两个有限自动机是等价的。注意，语义也可以定义为下面的函数 $g_{\Sigma}^{*}: I^{*} \rightarrow O^{*}$：

$$g_{\Sigma}^{*}(r) = r, g_{\Sigma}^{*}(tx) = g_{\Sigma}^{*}(t)\, g_{\Sigma}(tx)$$

也就是说，语义是从输入串到输出串的函数(映射)。容易得出 $g_{\Sigma}^{*}(t) = g_{\Sigma'}^{*}(t), t \in I^{*}$ 当且仅当 $g_{\Sigma}(t) = g_{\Sigma'}(t), t \in I^{+}$。

Moore 有限自动机和 Mealy 有限自动机类似，只是输出函数 γ 被 $\gamma': S \rightarrow O$ 所代替。也就是说，输出是和状态有关而不是和转换有关。可以把这样的输出看作是达到某一状态后触发的动作。可以为 Moore 有限自动机定义类似的语义，也就是对于某一输入串，有限自动机返回最终输出(字符)(g_{Σ} 函数)或是整个输出串(g_{Σ}^{*} 函数)。对于一个 Mealy 有限自动机 Σ 和一个 Moore 有限自动机 Σ'，如果对每一个可能的输入串(sequence)，Σ'的输出串恰好等于在 Σ 的输出串前加一个任意但固定的符号(Symbol)所形成的串(这个符号是 Moore 有限自动机在其初始状态的输出)，则说 Σ 和 Σ'是等价(或相似)的。在参考文献[2]中已证明：给定一个 Mealy 有限自动机 Σ，可以构造一个等价的 Moore 有限自动机 Σ'；反之亦然。

从上述形式化的定义，可以看到状态转换图(STD)是一个简单有向图，其中节点表示状态，表示在不同时刻的不同输入值结合；标有触发事件和条件的箭头表示转换，转换是由输入引起的状态转换。状态转化图可以帮助理解系统行为。下面看一个自动售货机(Vending Machine)的例子。

图 8-2 是一个销售饮料的自动售货机。如图上标识的，Bill 和 Coin 处可以分别投入纸币和硬币；Display 是一个液晶显示器，可以显示投入的金额或还需投入的数额；Button 处是多个按钮，可以选择不同的饮料；Change 处递出找回的零钱，而 Dispenser 处递出饮料。图 8-3 给出了自动售货机工作过程的状态转换图。

图 8-3 中圆角矩形表示状态，矩形内的文字为状态名称，描述了机器所处的状态；箭头表示状态间的转移，箭头上的文字表示触发状态转移的事件；另外有两个特殊表示的状态，实心圆表示状态图的初始状态 Sb，内含一个实心圆的圆圈表示状态图的终止状态 Se。从初始状态起，机器加电后(Power On)，处于空闲状态(S1)，等待顾客到来；若断电(Power Off)，则到达终止状态。顾客向机器中投钱后(Money Input)，机器显示钱数(S2)，若投入钱数足够(Enough Money)，则机器递出碳酸饮料(S4)，投入超额的话还要找回零钱；若钱数不够(Money not enough)，机器则显示差额(S3)，并等待顾客继续投钱，直到投入的钱数足够，才会递出碳酸饮料并找零。机器卖出碳酸饮料后自动回到空闲状态。在机器成功卖出碳酸饮料之前，如果顾客取消交易(Cancel)，则机器退回顾客已经投入的钱(S5)，并回到空闲状态。如果顾客投入假币(Fake Money)，机器会立即退回，并返回空闲状态。

根据定义 8.1，可以写出图 8-3 对应的六元组$(S, I, O, \delta, \gamma, s_0)$：

- $S = \{S1, S2, S3, S4, S5\}$。

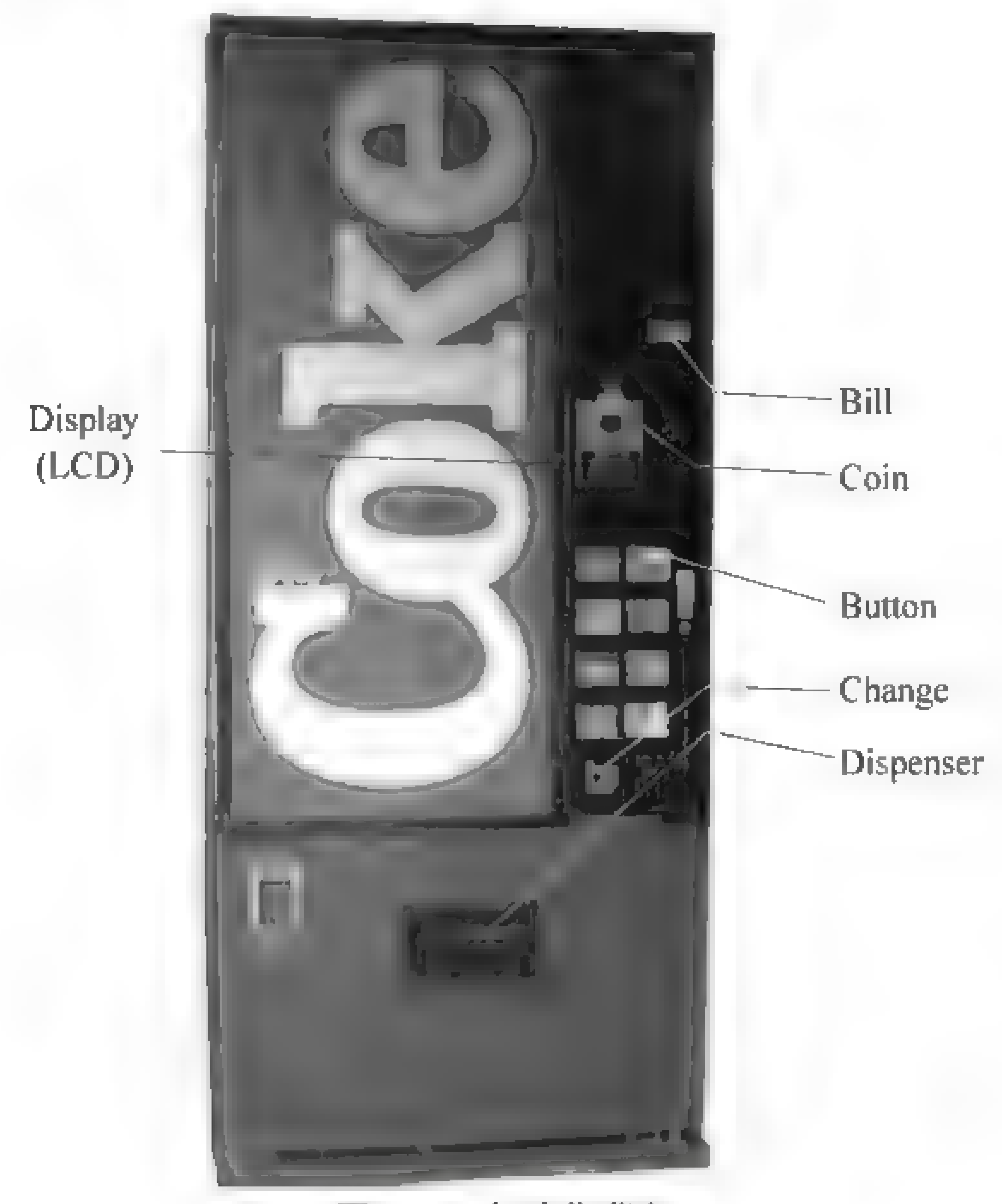


图 8-2 自动售货机

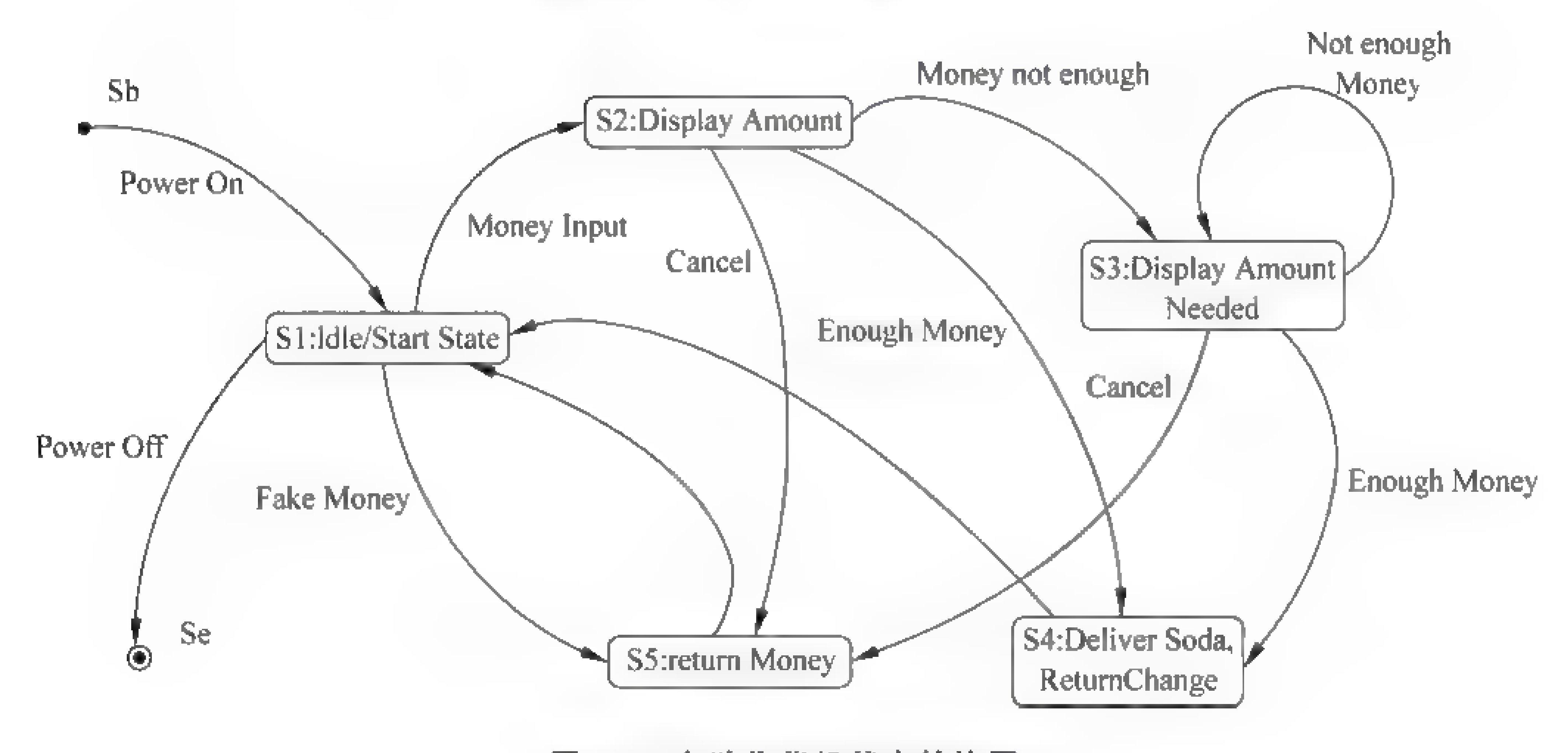


图 8-3 自动售货机状态转换图

- $I=\{EM,NEM,CL,FM,\varepsilon\}$，其中，EM 表示足额的钱，NEM 表示不足额的钱，CL 表示取消操作，FM 表示假币，ε 表示空输入。
- $O=\{Soda,Change,FM\}$，其中，Soda 表示碳酸水，Change 表示零钱，FM 表示假币。
- $\delta=\{\delta_i \mid 1\leqslant i\leqslant 11\}$，$\delta_i$ 列举如下：

$\delta_1(S1,EM)=S2, \delta_2(S1,NEM)=S2, \delta_3(S1,FM)=S5,$

$\delta_4(S2,EM)=S4, \delta_5(S2,NEM)=S3, \delta_6(S2,CL)=S5,$

$\delta_7(S3,EM)=S4, \delta_8(S3,NEM)=S3, \delta_9(S3,CL)=S5,$

$\delta_{10}(S4,\varepsilon)=S1, \delta_{11}(S5,\varepsilon)=S1$

- $\gamma=\{\gamma_i \mid 1\leqslant i\leqslant 5\}$，$\gamma_i$ 列举如下：

  $\gamma_1(S1,FM)=FM, \gamma_2(S2,EM)=Soda, \gamma_3(S3,EM)=Soda,$

  $\gamma_4(S2,CL)=Change, \gamma_5(S3,CL)=Change,$

- $s_0=Sb$。

8.2 状态图

状态转换图无层次结构，状态数目与转移数目随着系统复杂性增加而呈指数增加趋势，导致状态杂乱、难以理解。为了解决 STD 图的缺陷，Harel 提出了状态图(Statechart)，Harel 的状态图是状态转换图、有限状态机的扩展。状态图表示系统行为，提供可视化形式，以模块化风格描述其状态和转移。状态图用来控制和组织详细信息，如果这些信息用表格形式表示，则所能提供的清晰度将会降低。状态图允许超级状态有历史信息，当系统崩溃时，历史信息在系统恢复方面是很有用的。

图 8-4(a)是一个简单状态转换图。在状态 U 时，如果事件 E 触发，则转移到状态 S；如果事件 H 触发，则执行动作 C(H/C 表示事件 H 发生时采取行动 C)，转移到状态 T。在状态 S 时，如果事件 F 触发，则转移到状态 U；如果事件 G 触发，则执行动作 A，转移到状态 T。在状态 T 时，如果事件 F 触发，则转移到状态 U；如果事件 E 触发，则执行动作 B，转移到状态 S。我们注意到，从状态 S 和状态 T，当事件 F 触发，则发生两个转移，都转移到状态 U。如果把状态 S 和状态 T 合并或结群(Cluster)成一个超级状态 D，那么从超级状态 D 到状态 U 的转移只有一个，如图 8-4(b)所示。图 8-4(b)是一个状态图，与 STD 相比减少了转换的数量。

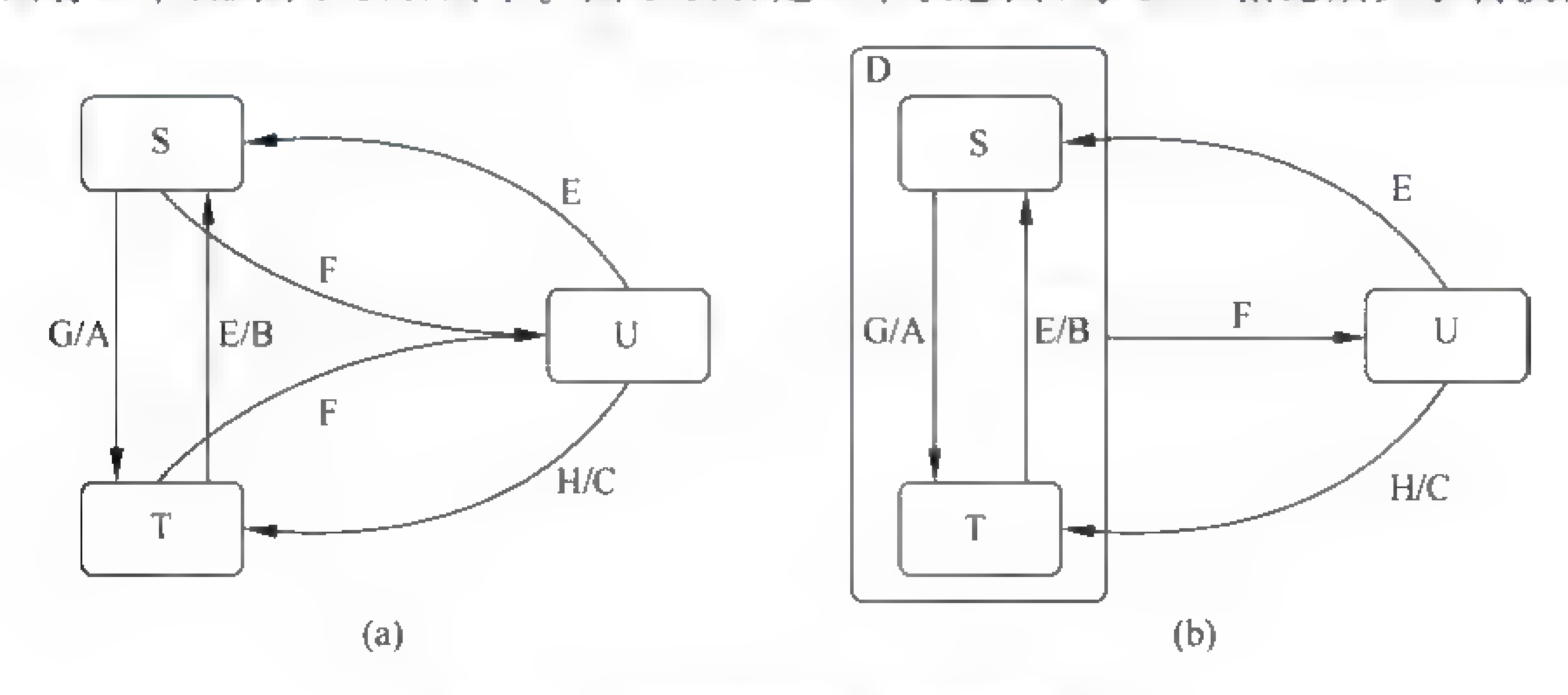


图 8-4 状态转换图(a)与状态图(b)

由状态 S 和状态 T 合成的超级状态 D 被称作“或”状态，即任一时刻超级状态 D 只处于它的某一个子状态，状态 S 或状态 T，而不能同时处于两种状态。这称为状态图的“或”属性。

8.2.1 Harel 状态图的属性

状态图提供层次结构可减少系统建模复杂性，支持内容抽象、并发、正交性、全局通信机制、历史状态，具有简洁性、表达能力强，可进行状态的“与”/“或”(AND/OR) 分解。下面介绍这几种状态图属性。

1. “或(OR)” 状态

一个超态可以分解为任意多个 OR 子状态，当一个对象处于超态时，它必须处于其中的一个而且是唯一的一个“或”子状态。在图 8-5 中，超态 S 分解为两个子状态 U 和 V。S 被称作“或”状态，U 和 V 是其两个“或”子状态。任何时候 S 或处于 U，或处于 V，而不能同时处于两个状态。U 是进入 S 的默认状态。

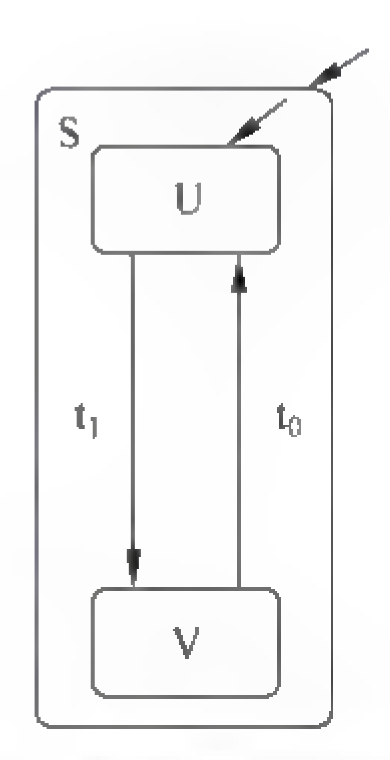


图 8-5 “或”状态与“或”子状态

2. “与(AND)”状态

一个超态可以分解为任意多个 AND 子状态，当一个对象处于超态时，它必须处于每一个活性的“与”子状态(对象正处于的状态称为活性状态)。在图 8-6 中，超态 C 分解为两个子状态 A 和 B。C 被称作“与”状态，A 和 B 是其两个“与”子状态，由点划线分割。要注意的是，A 状态相对于 X、Y、Z 状态是“或”状态；B 状态相对于 R、S 状态是“或”状态。任何时候 C 将同时处于 A 的一个子状态及 B 的一个子状态。

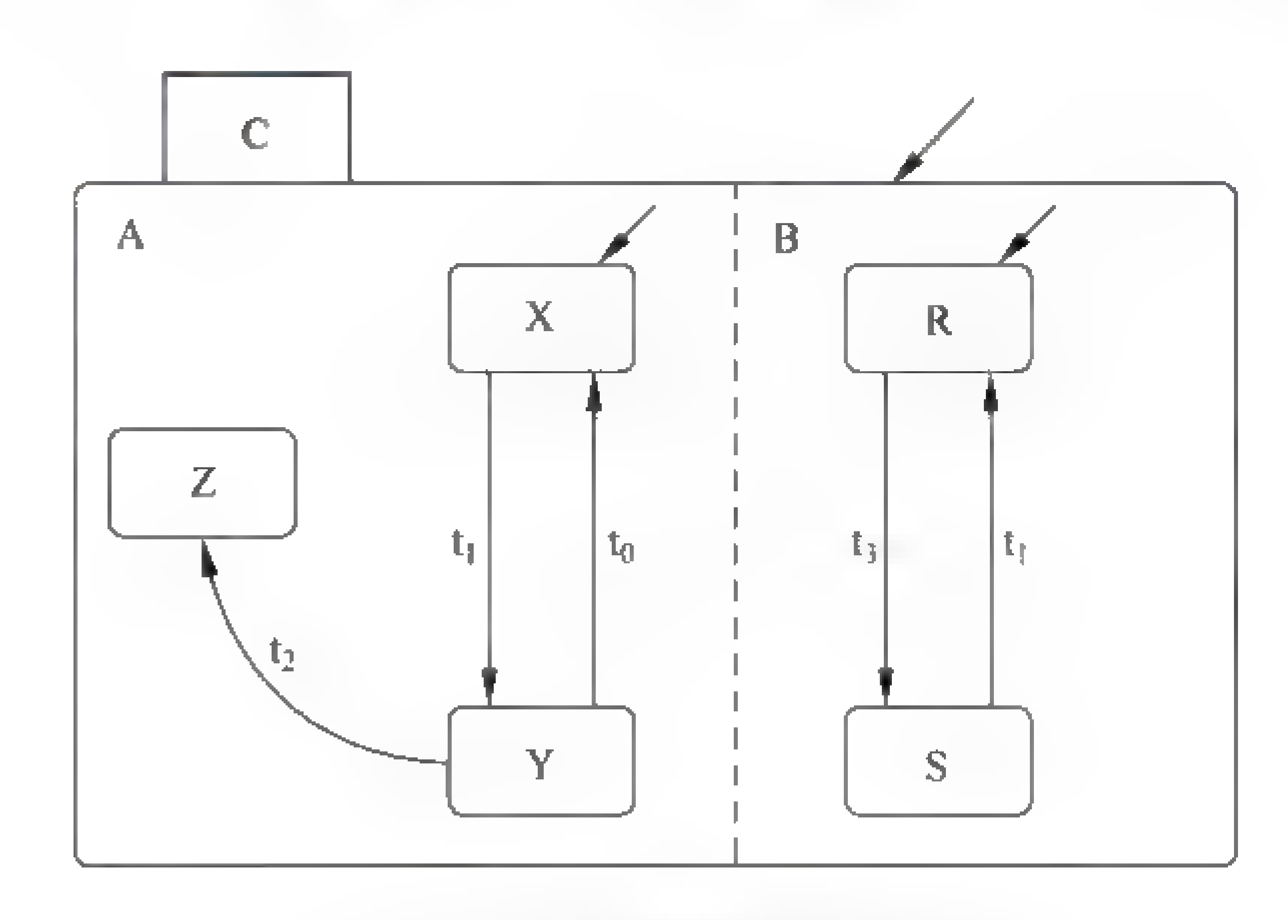


图 8-6 “与”状态与“和”子状态

3. 聚类与细化

聚类(Clustering)是自底向上的概念,而细化(Refining)是自顶向下的概念。二者都描述一个状态与其子状态的关系。聚类减少了状态图中转换的数量。为了看清状态转换内部的细节情况,超态可以按照需要分解、展开,这个过程称为细化。在图8-7(a)中,聚类状态S和状态T得到图8-7(b)中的超态D,从而减少了一个F转换;为了明确超态D内部状态转换情况以及确认转换E与H/C的目标状态,把D状态分解、细化成S和T状态,如图8-7(c)所示,于是问题变得很明了。

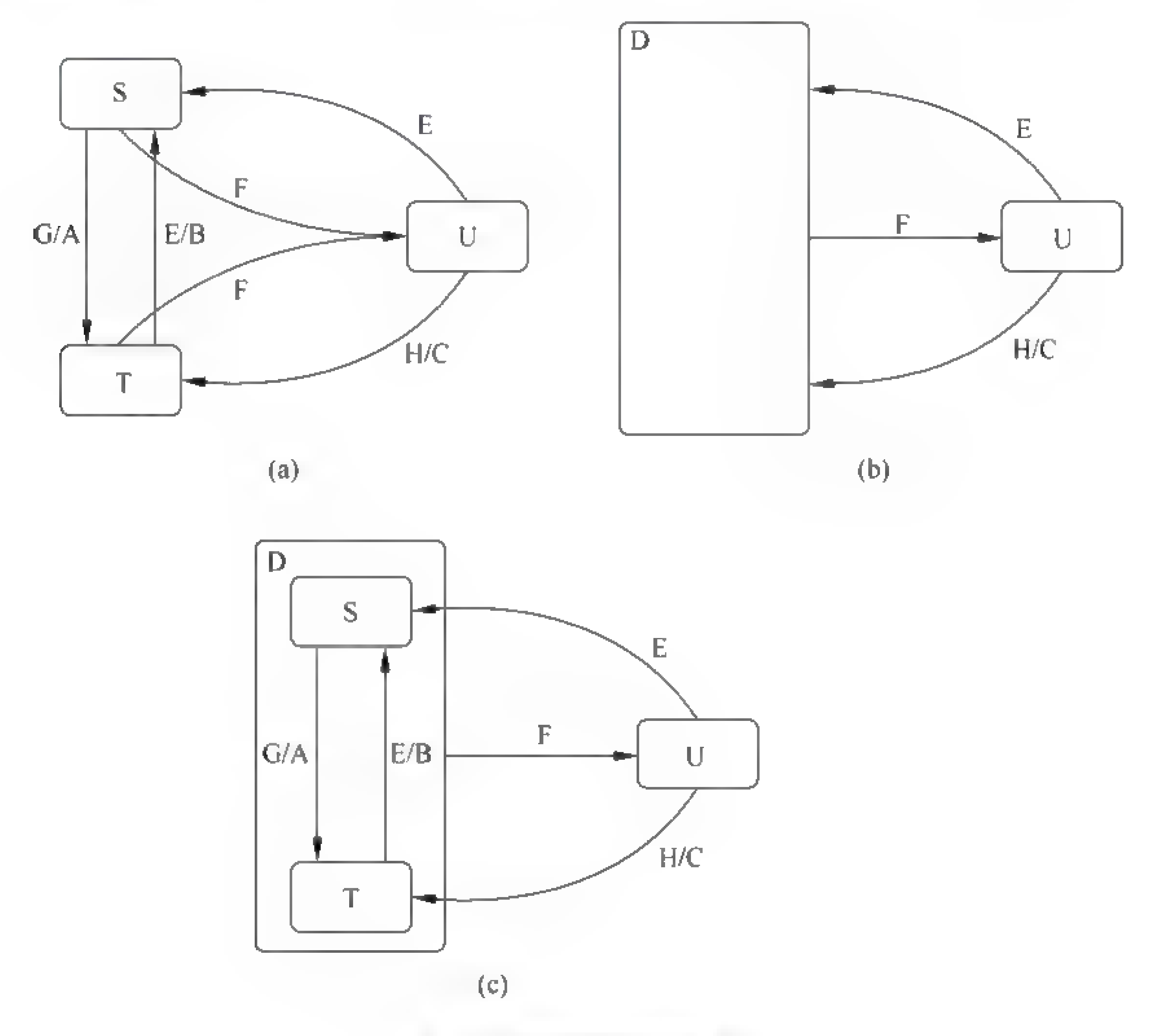


图8-7 状态聚类与细化

4. 历史态

状态图中的历史态(History State)给出了超态最近被访问的状态。历史态分为"浅"历史态与"深"历史态。图8-8(a)中的H为"浅"历史态;图8-8(b)中的H* 为"深"历史态。简单地说,"浅"历史态H是表示最近进入的并与其处于同级的状态,而"深"历史态H* 表示最近访问的处于任意深度级别上的子状态。

如图8-8(a)所示,当状态图退出超态K时,历史态H记录与其处于同一级的当时的活

性状态G或是F，但不是两者同时；当系统返回超态K时，根据历史态H记录，使G或F成为活性态。如G成为活性态，根据它的默认状态，超态K处于B状态；如F成为活性态，根据它的默认状态，超态K处于C状态。

如图8-8(b)所示，当状态图退出超态K时，“深”历史态H*记录当时超态K处于一个深度子状态，或是G的一个子状态或是F的一个子状态，但不是两者同时；当系统返回超态K时，根据历史态H*记录，使G或F成为活性态。如G成为活性态，根据“深”历史态H*记录的最近访问情况，超态K可能处于A状态，而不是根据G的默认状态，使超态K处于B状态；如F成为活性态，超态K可能处于D或E状态，而不是根据F的默认状态，使超态K处于C状态。

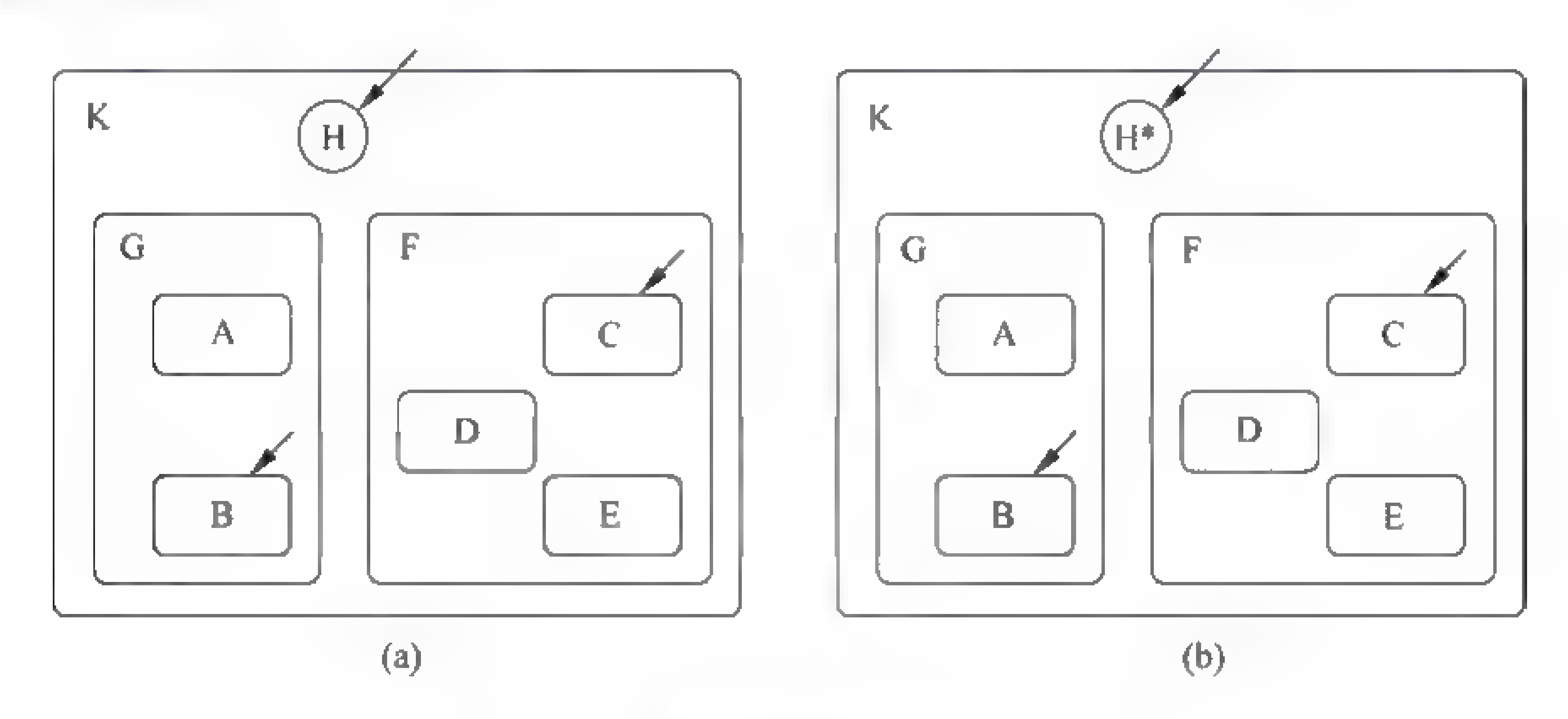


图8-8 具有历史状态的状态图

5. 正交性(Orthogonality)

正交性本质上是一个AND分解，但注重描述AND状态之间或组件之间的同期并发情况。图8-9(a)中的Y状态由两个正交组件组成：A与D，由AND运算关联着。处于状态Y等价于既处于状态A又处于状态D。图8-9(b)是图8-9(a)等价的“平面”版，代表一种状态机乘积，给出了图8-9(a)所要表述的语义(8.2.2节中将进一步讨论由状态图到状态转换图的变换)。注意：在图8-9(a)中，当Y状态处于状态构造(B,F)时，事件e的出现，产生状态“转移同时性”，即A里从B转移到C，D里从F转移到G，使得Y状态处于状态构造(C,G)；当事件p的出现，无论Y状态处于何种状态构造，都将从Y状态转移到I状态；处于I状态时如果事件e的出现，将从I状态转移到Y的状态构造(C,G)，此种情况称为“事件分裂”；处于Y状态的状态构造(C,E)时，如果事件n出现，将从Y状态转移到H状态，此种情况称为“事件合并”。

另外应注意，在图8-9(a)中，一个特殊条件[in(G)]附在从A的子状态C的“f转移”上，即事件f出现时，A从子状态C转移到子状态B的条件是D处于子状态G，这一点在图8-9(b)得到了反映。

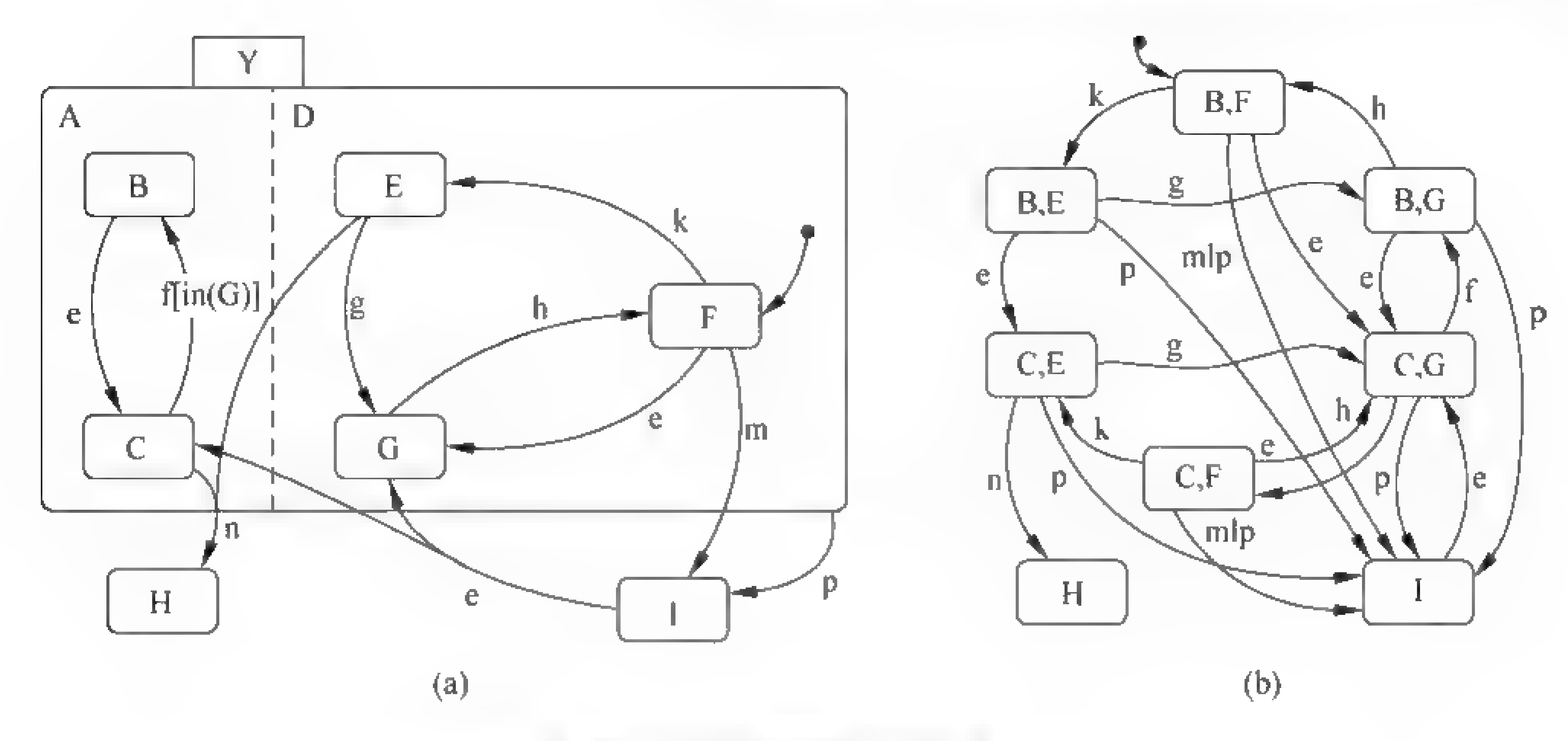


图 8-9 状态图中的正交性

图 8-9 展示出状态指数爆炸问题的核心，即 Y 的“显性”版（也称“平面”版）的状态数目是其各“和”子状态数目的乘积。正交组件之间的同步可以通过响应共同事件（如图 8-9 中的事件 e），相互影响可以使用特殊条件[in(state)]。除此之外，建立状态图的并发模型还可以容许增添“输出”事件。“输出”事件也可称为“行动”，在转移上可有选择地把行动添附于触发事件上。

此行动不仅作用于外界，正交组件之间也可利用行动相互作用。这可以用事件广播（Broadcast）机制来实现：正像一个外部事件的出现会引起所有组件内相关转移发生一样，如果事件 m 出现并且标有 m/e 的发生，则行动 e 马上被激活，并被看作一个新的事件出现，可能引起其他组件内进一步转移发生。如图 8-10 所示，当 H 组件处于 J 状态时，事件 m 出现会激发转移 m/e 的发生，并发出行动 e，如果 A 组件和 D 组件各自分别处于 B 和 F 状态，

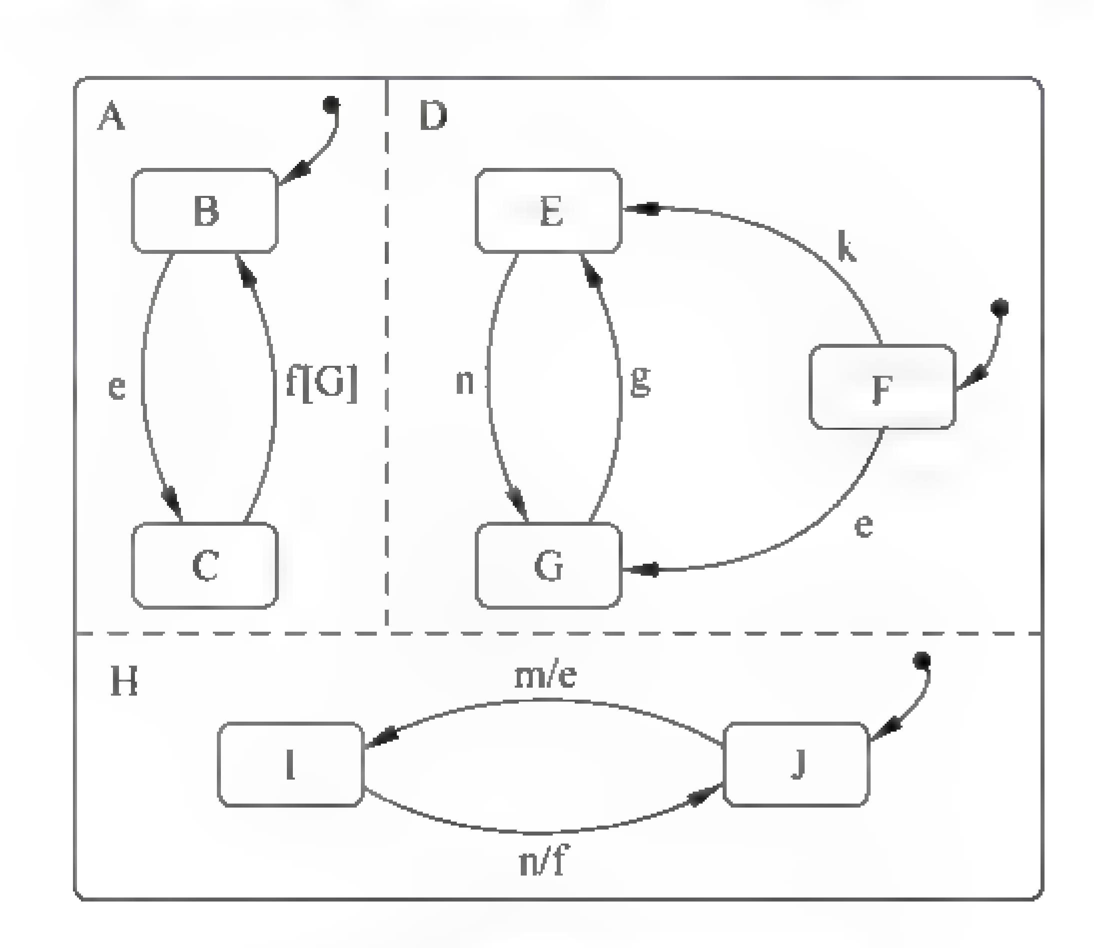


图 8-10 状态图中的事件广播

则在 e 的作用下将会发生相应的转移，这种情况称事件 m 发生的作用长度为 2。当 H 组件处于 I 状态时，事件 n 的出现可能引起的作用长度为 3：H 组件从 I 状态转移到 J 状态，并发出事件 f；如果 D 组件处于处于 E 状态，则转移到 G 状态；如果 A 组件处于 C 状态，则转移到 B 状态。

8.2.2　从状态图变换到 STD

为了导出测试用例，需将层次化的状态图变换为"平面"版的状态转换图。由图 8-9(a)～图 8-9(b)表示了这种变换过程。为了说明方便，可用图 8-11 中的简洁状态图来描述其变换步骤。

图 8-11(a)是一个状态图，超态 R 和超态 T 是正交关系；图 8-11(b)是由图 8-11(a)变换的状态转换图。从图 8-11(a)中的状态 A 开始，它对应于图 8-11(b)中的状态 A。当事件 p 出现时，图 8-11(a)从状态 A"同时"分别转移到超态 R 的 U 状态和超态 T 的 X 状态；在图 8-11(b)中从状态 A 转移到"U，X" 状态。此时如果事件 j 出现时，图 8-11(a)中超态 R 的 U 状态不响应事件 j，保持原状态，超态 T 的 X 状态响应事件 j，转移到 W 状态，在图 8-11(b)中从状态"U，X"转移到"U，W"状态；如果事件 e 出现时，图 8-11(a)中超态 R 从 U 状态转移到 V 状态时，超态 T 从 X 状态转移到 Y 状态，在图 8-11(b)中从状态"U，X"转移到"V，Y"状态。图 8-11(a)中当超态 R 处于 V 状态时，如果事件 f 出现而且超态 T 处于 Y 状态，即满足条件[in(Y)]，超态 R 从 V 状态转移到 U 状态，超态 T 仍处于 Y 状态，在图 8-11(b)中从状态"V，Y"转移到"U，Y"状态。对于事件 g、k 的出现以及引起的状态转移可以用同样方式分析得到，留给读者作为练习。

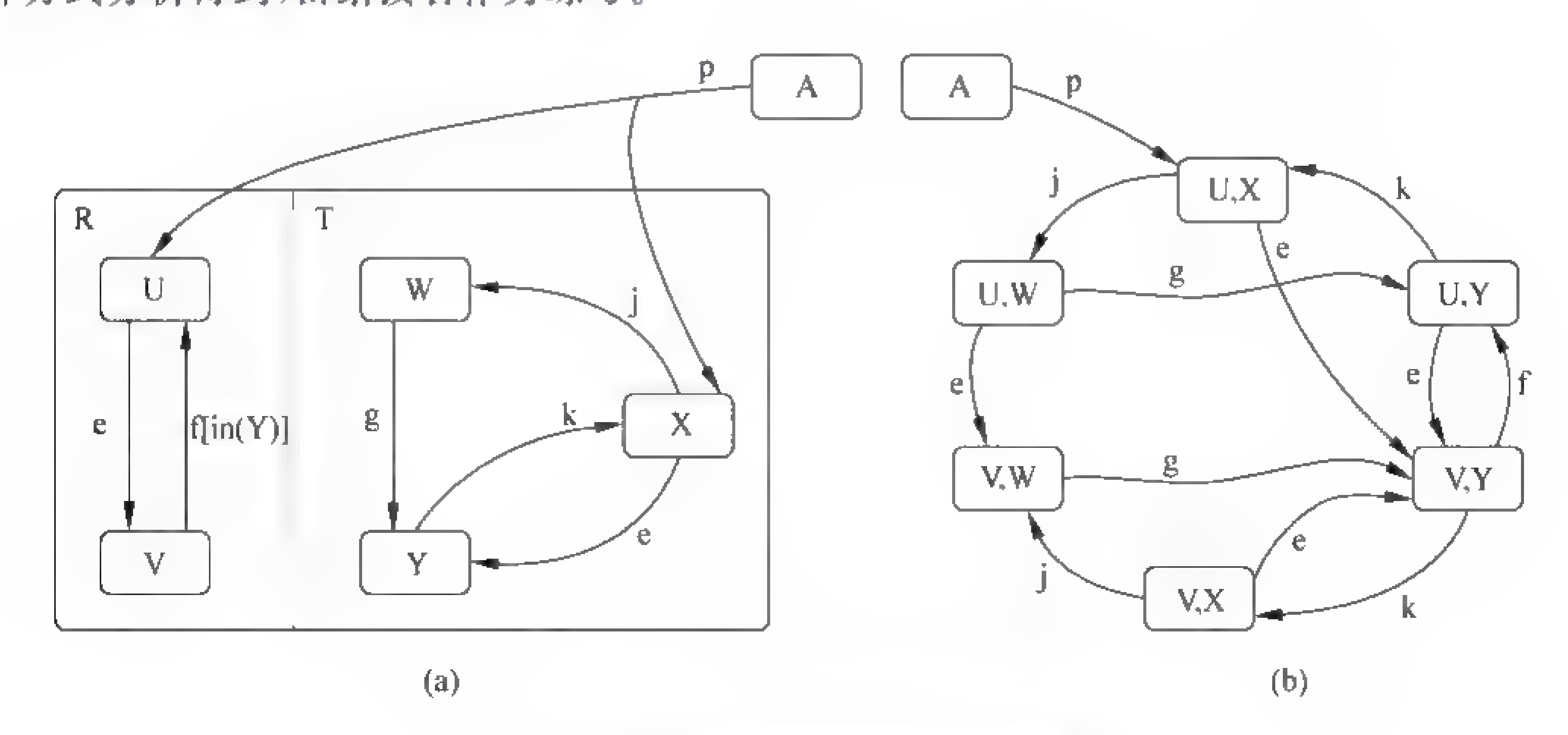


图 8-11　状态图变换到状态转换图

8.2.3 UML状态图

David Harel发明的状态图最初是为面向功能的系统开发的，后来做少量修改后用在面向对象的系统。UML状态图为对象管理服务，是在Harel的状态图的基础上扩展了一些新特征。在UML的状态图(Statechart)中，根据变量所具有的值以及值的数据类型，对象可能处于不同的状态。表8-1给出了Harel状态图和UML状态图的属性比较。

表8-1 比较Harel状态图和UML状态图的属性

属性	Harel状态图	UML状态图
内嵌与正交状态	支持	支持
单个转移代表从不同子状态的相同的事件	支持	支持
事件广播	支持	支持
历史状态	支持	支持
子状态机	支持	支持
重叠状态(一个子状态属于一个以上的父状态)	支持	无
伪状态(代表转换路径中的瞬间点)	无，但其连接符进行同样操作	支持
Fork(分叉)与Join(合并)方法	用Fork与Merge(合并)	用伪状态实现
事件可以带有参数	无	支持
自由转移	当一个退出转移离开组合边界时，发生不一致情况	通过定义一个自由边界退出转移，防止了自由转移
实现同期的方法	有限的方法：用IS_IN参数；广播通信	多种方法：广播通信，Fork(分叉)，Join(合并)，利用被传播的事件，用IS_IN操作，用synch伪状态
对于事件处理	由最外层状态机负责	由最内层状态机负责

8.3 基于状态的测试

基于状态的测试一般是用状态图来描述事件序列，或称为用例场景，并由此产生测试用例。白盒测试技术是以代码覆盖为标准来决定测试用例产生数量、测试结束标准；基于状态的测试评价标准是状态、转移覆盖及对于不正常、不相关事件的考虑，并以此决定测试用例产生数量和测试结束条件。

8.3.1 测试步骤

利用图形化技术进行测试的一般步骤可用图 8-12(a)表示，而基于状态图的测试是其一个实例，用图 8-12(b)表示。

1. 创建图形化规格说明书

需求的图形化表示模型包括数据流图(DFD)、实体关系图(ERD)、状态图(Statechart)、状态转化图(STD)、对话图和类图，它们可以作为需求分析工具。用这些图对问题域进行建模，对于复杂的系统行为、语义进行描述，或者用于创建新系统的概念表示法。另外，图形有助于分析者和客户在需求方面形成一致的、综合的理解，可以发现需求的错误。图形化技术在创建规格说明书中得到广泛应用，其结果称之为图形化规格说明书。层次化状态图可以用来帮助描述系统的规格(如图 8-12(b)所示)。

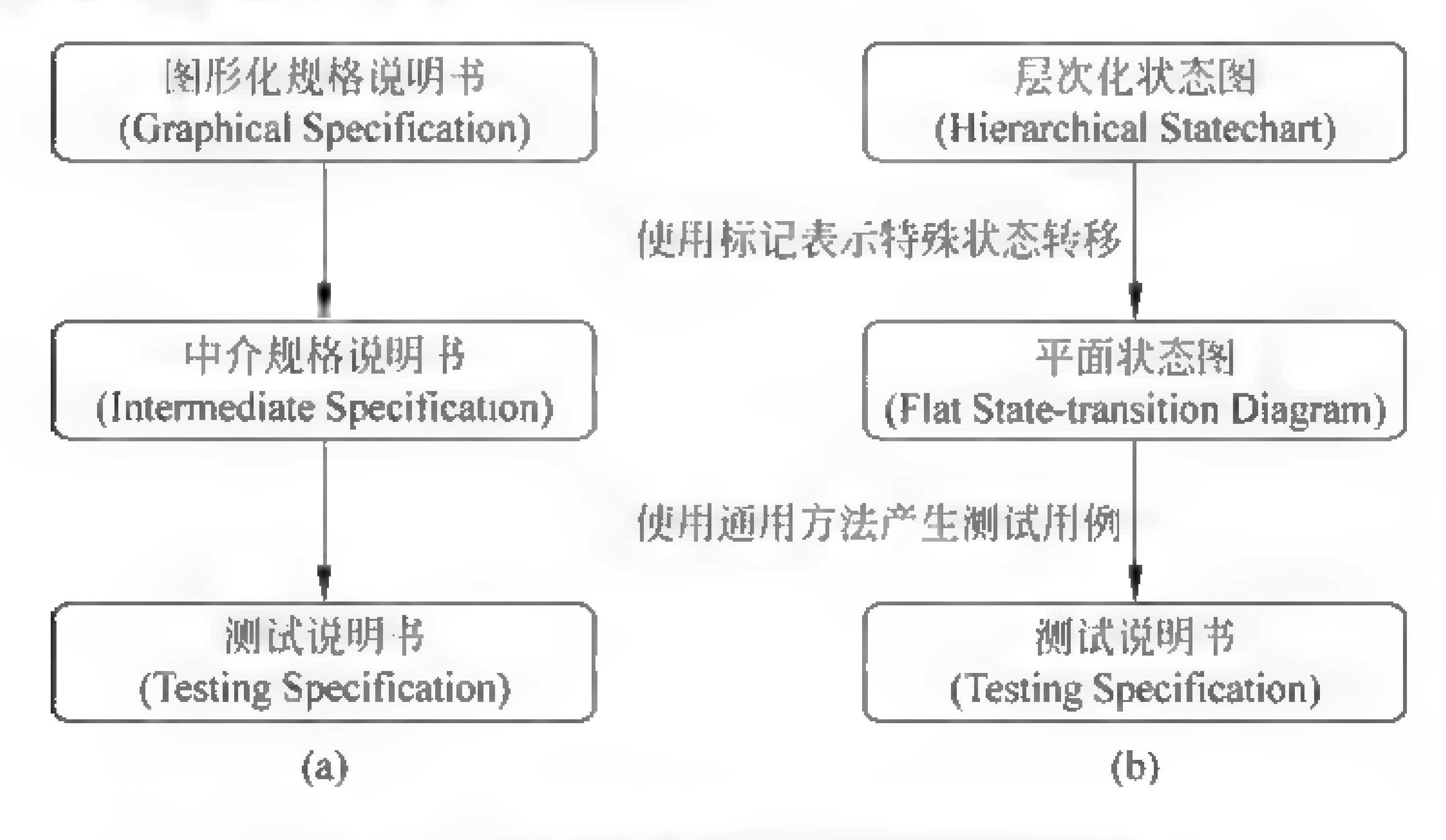


图 8-12　利用图形化技术进行测试的步骤

2. 产生中介规格说明书

选用图形化规格说明书的目的是要方便生成测试说明书。其中状态转移图和状态图是两种常用的图形化技术。考虑到测试不正常及不相关事件，可能需要在状态转化图添加特殊状态转移标记；对于层次化状态图，需要将其转换成状态转移图即平面化(如图 8-12(b)所示)。由此类修改得到的结果称之为中介规格说明书。“平面化”的状态转移图是层次化状态图的中介规格，可以用来生成测试用例。

3. 生成测试规格说明书

上述中介规格说明书是一种有向图。其中一个节点为始点(可能是哑节点)，一个节点为终点(也可能是哑节点)。遍历从始点到终点的所有路径，包括有效路径及无效路径。每

条路径对应于一个测试序列。有效路径是指对于一系列有效输入，系统相应的一系列响应；无效路径是指对于一系列无效输入，系统相应的一系列“意外”处理。可遍历的路径数目会很多甚至是无限的。在这种情况下，需要确定某种路径生成规则。

8.3.2　产生测试用例

按照上节所讲述的测试步骤，可以由状态转换图、状态图产生测试用例。由状态转换图产生测试用例，实际上是遍历一个有向图，从始点到终点的每一条路径代表一个测试场景，从每一个测试场景可以产生一个或多个测试用例。如果是从层次化状态图开始，第一步是将其转换成“平面”状态转换图，其过程与方法在 8.2.2 节中(图 8-11)介绍过，下面用一个 ATM 的例子讲述从状态转换图产生测试用例的过程。

1. ATM 系统需求

所要实现的软件系统是要控制一个模拟的 ATM——自动取款机。ATM 装有磁性条码读取器用于读取 ATM 卡上信息，键盘与显示屏用于和用户交互，存入现金入口，还有领取现金出口。ATM 机通过合适的网络连接与银行通信。

ATM 机每次给一个用户提供服务。用户需插入 ATM 卡，输入个人身份号码(PIN)。作为处理事务的一部分，卡号与个人身份号码被送到银行进行确认；然后用户可以执行一个或多个事务处理。ATM 机一直保留 ATM 卡直到用户指示退出系统操作，此时将卡退还给用户。

ATM 机必须能为用户提供以下服务：

(1) 用户必须能从 ATM 卡的任一有效账户上提取现金，提取的金额是 $20.00 的整数倍，每次现金支付时必须得到银行的认可。

(2) 用户必须能在 ATM 卡的任一有效账户上存款，指放入信封里的现金或支票。用户向 ATM 输入存款数额，银行操作人员收到信封后要手工核对数额。每次存款时银行必须收到物理形式的信封后才加以认可。

(3) 用户必须能在 ATM 卡的任两个有效账户之间进行货币转账。

(4) 用户必须能查询 ATM 卡的任一有效账户上的存款余额。

ATM 机每次交互都通知银行以获得银行的验证。在提取现金或存款的情况下，当事务完成后(即现金支付后或信封收到后)，要再发送一个消息给用户以示确认。

如果银行确认用户的 PIN 无效，在事务进行之前，要求用户再输入 PIN。如果用户输入 3 次都不成功，ATM 机将永久地保留 ATM 卡，用户必须与银行联系方可取回 ATM 卡。如果不是因为无效 PIN 而是其他原因，ATM 机将显示对于问题的解释，并问用户是否要进行另一项事务交互。对于每一个成功的事务处理，ATM 机给用户打印一个收据，提示日期、时间、ATM 机位置、交互类型、账户、数额、转出与转入账户余额。

ATM 机有一个带有钥匙操作开关面板，安置在银行内部，让银行操作员启动或停止用户服务。当开关置于 off 位置时机器关闭，操作员可以取下存入的信封，给机器装入现金、空白收据等。从开关面板启动系统之前，操作员验证并放入手头的现金总额。

2. 由系统需求绘出状态图

根据上述 ATM 需求描述，对系统进行抽象得到图 8-13。所谓"抽象"是指将我们"所关心的"系统行为用一种形式表示出来，而省去与分析不相关或内部细节部分。图 8-13 为描述以上系统需求的状态图，其中圆角矩形表示状态，矩形内的文字为状态名称；箭头表示状态间的转移，箭头上的文字表示触发状态转移的事件；另外有两个特殊表示的状态，实心圆表示状态图的初始状态，内含一个实心圆的圆圈表示状态图的终止状态。

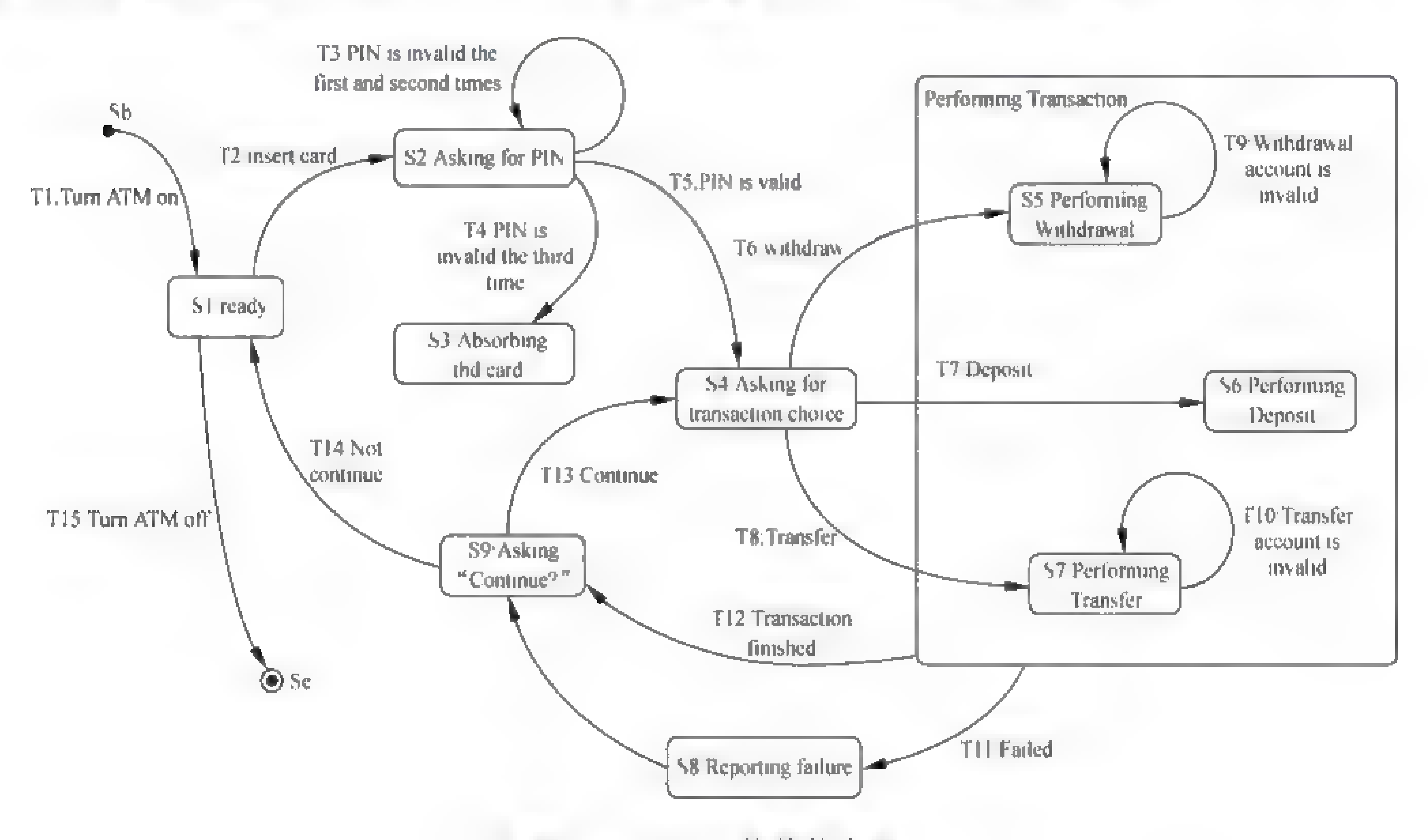


图 8-13 ATM 软件状态图

从图 8-13 中，可以得到主要的系统需求。从初始状态开始(Sb)，银行操作员启动 ATM 服务(T1)，使 ATM 处于 ready 状态(S1)；若在 ready 状态关闭服务(T15)，则到达终止状态(Se)。在 ready 状态，用户插入 ATM 卡(T2)，ATM 转到 Asking for PIN 状态(S2)，要求用户输入 PIN。如果输入的 PIN 有效(T5)，ATM 转到 Asking for transaction choice 状态(S4)，要求用户选择事务类型；如果 PIN 无效(T3)，则 ATM 仍处在 Asking for PIN 状态，要求用户重新输入；如果连续 3 次输入无效 PIN(T4)，ATM 将没收 ATM 卡。在 Asking for transaction choice 状态，用户可以选择取款(T6)，存款(T7)和转账(T8)3 种事务，ATM 会分别转移到 Performing Withdrawal(S5)、Performing Deposit(S6)和 Performing Transfer 状态(S7)，要求用户输入金额(S5 的有效金额为 $20 的整数倍且不超

过账户余额，S7 的有效金额不能超过账户余额）。

如果用户输入无效金额（T9 或 T10），则 ATM 转回原状态，要求用户重新输入。当任何一种事务成功完成时（12），ATM 转移到 Asking "Continue?"状态（S9），询问用户是否继续操作；若事务失败（T11），则 ATM 会先转到 Reporting failure 状态（S8），报告错误信息，然后再转到 Asking"Continue?"状态。如果用户选择继续操作（T13），ATM 会又回到 Asking for transaction choice 状态，要求用户选择事务类型；否则（T14），ATM 回到 ready 状态，等待下一个用户到来。

3. 由状态转换图产生测试场景

首先回顾一下上节讲述的测试步骤：

(1) 创建图形化需求规格说明书，即由系统需求绘出层次化状态图。

(2) 产生中介规格说明书，即由层次化状态图导出平面化状态图。

(3) 产生测试规格说明书，即遍历平面化状态图中所有从始点到终点的路径，每一条路径代表一个测试场景。

(4) 产生测试用例，即对每一个测试场景，输入不同的测试数据，从而形成多个测试用例。

图 8-13 虽然是一个层次状态图，但是其中只有一个"或"状态 Performing Transaction，而且已经标出了到其 3 个子状态的转移路线。图 8-13 的层次信息很简单，所以为了节省篇幅，省略了对上述步骤(2)的描述，直接把图 8-13 作为一个平面化的状态转换图，从中产生所有的测试场景。

图 8-14 是从图 8-13 中抽取的一条路径，代表一个测试场景（TS1），称为 Test Senario。该路径描述如下：

TS1：Sb • T1 • S1 • T2 • S2 • T5 • S4 • T6 • S5 • T9 • S5 • T12 • S9 • T14

类似地，把图 8-14 中的 S5 状态替换成图 8-13 中的 S6 或 S7 状态，可以得到测试场景 TS2 和 TS3。分别描述如下：

TS2：Sb • T1 • S1 • T2 • S2 • T5 • S4 • T7 • S6 • T12 • S9 • T14

TS3：Sb • T1 • S1 • T2 • S2 • T5 • S4 • T8 • S7 • T10 • S7 • T12 • S9 • T14

再考虑一条无效路径，如图 8-15 所示，描述如下：

TS4：Sb • T1 • S1 • T2 • S2 • T3 • S2 • T3 • S2 • T4 • S3

4. 由测试场景产生测试用例

由测试场景产生测试用例，就是在需要输入数据的步骤输入一系列各不相同的测试值，来检验在各种情况下系统是否满足需求。下面针对 TS1 产生测试用例，以便说明其过程。

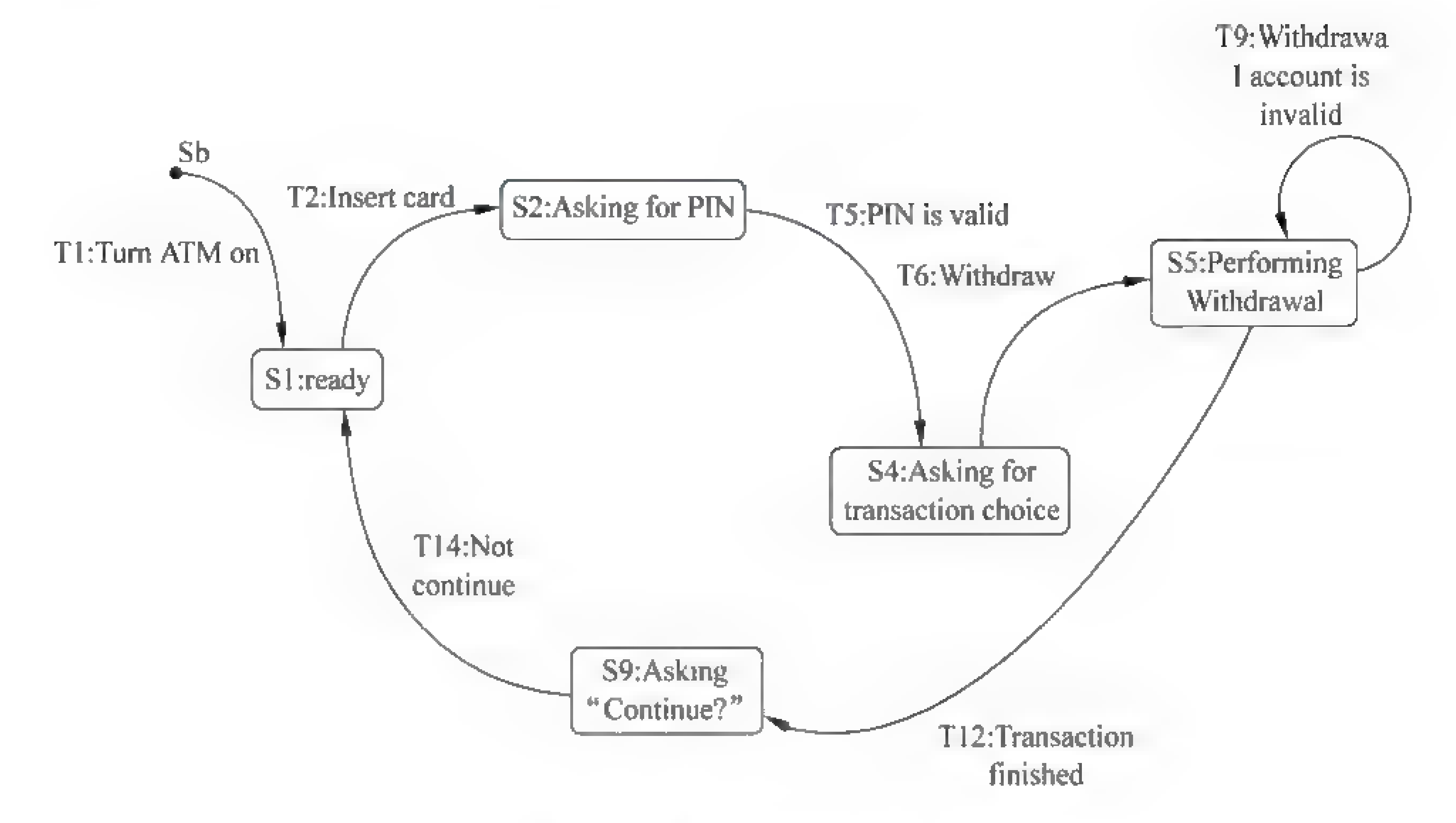


图 8-14　测试场景 TS1

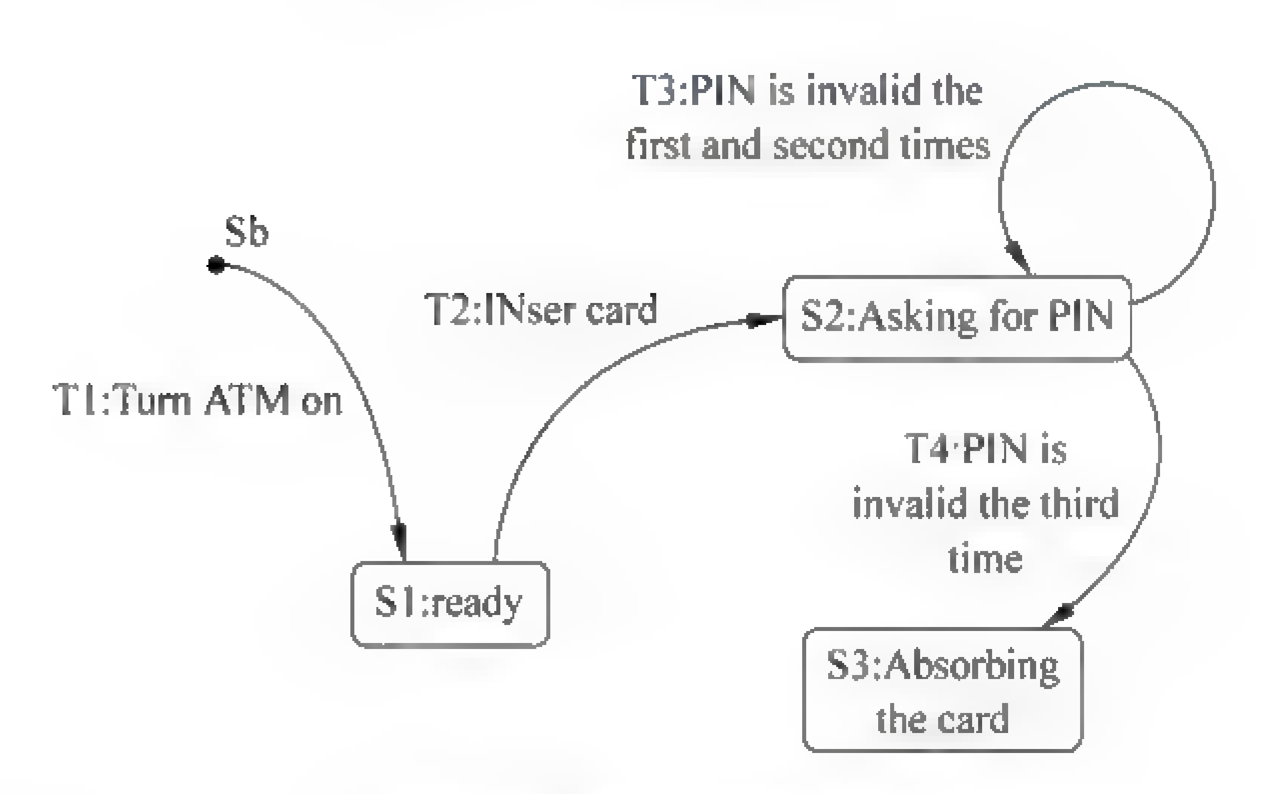


图 8-15　测试场景 TS4

首先假设银行的数据库中存有如表 8-2 所示的用户信息。

表 8-2　数据库账户表

账号	PIN	账户余额
977764435433543	452765	$ 5760
977763436571288	332456	$ 355.56

使用账户 977764435433543，产生的 3 个测试用例(TC1、TC2、TC3)列在表 8 3 中(注：表 8 2 中 SC 表示 ATM 显示的界面)。

表 8-3 TS1 产生的测试用例表

测试场景	测试用例	输入	输出(系统响应)	输出说明
TS1	TC1		SC1	欢迎界面
		插卡	SC2	要求用户输入 PIN
		452765	SC3	要求用户选择服务
		选择“取款”服务	SC4	要求用户输入取款金额
		0	SC4	提示输入无效,要求重新输入
		20	SC5	询问是否继续
		选择“否”	SC1	回到欢迎界面
	TC2		SC1	欢迎界面
		插卡	SC2	要求用户输入 PIN
		452765	SC3	要求用户选择服务
		选择“取款”服务	SC4	要求用户输入取款金额
		635	SC4	提示输入无效,要求重新输入
		5760	SC5	询问是否继续
		选择“否”	SC1	回到欢迎界面
	TC3		SC1	欢迎界面
		插卡	SC2	要求用户输入 PIN
		452765	SC3	要求用户选择服务
		选择“取款”服务	SC4	要求用户输入取款金额
		6000	SC4	提示输入无效,要求重新输入
		1460	SC5	询问是否继续
		选择“否”	SC1	回到欢迎界面

8.3.3 覆盖分析

编码覆盖是白盒测试技术的重要组成部分,它提供选择产生测试用例技术的依据,决定测试结束条件。例如,如果用户关心路径覆盖,则可以选择“基本路径”测试技术产生测试用例,根据覆盖率决定产生测试用例的数目,而执行完所产生的测试用例,则是测试结束条件。基于状态测试的覆盖分析,是根据状态机或状态图相关元素或其组合来构造覆盖率,如状态覆盖率、转移覆盖率、状态-事件覆盖率等。这些覆盖率决定产生测试用例方法和数目,以及测试结束条件。

本节将介绍 4 种与基于状态的软件测试有关的基本覆盖类型(或称度量类型),包括状态覆盖、转移覆盖、事件覆盖和状态-事件覆盖。

(1) 状态覆盖：被覆盖的状态数目占给定状态模型里所有状态数目的比例。

(2) 转移覆盖：被执行的转移数目占给定状态模型里所有转移数目的比例。

(3) 事件覆盖：被覆盖的事件数目占给定状态模型里所有相关事件数目的比例。

(4) 状态-事件覆盖：被执行的状态-事件组合数目占给定状态模型里所有状态-事件组

合数目的比例。

状态覆盖要计算状态图遍历路径所覆盖的状态。有的状态可能被重复访问，但只计算一次。所访问的不同状态数目越多，状态覆盖率越高。但状态覆盖率高并不意味着其他覆盖度量效果好。如图 8-16 所示，遍历路径 S_1 T_2 S_2 覆盖所有状态，S_1 和 S_2 使状态覆盖率达到 100%，但转移覆盖率只达到 25%，即只涵盖转移 T_2，其他转移 T_1、T_3、T_4 没有涵盖到。

对于一个交互系统，我们关心此系统是否能响应或处理所有可能发生的事件。对此可以选择事件覆盖进行度量。事件覆盖要计算事件出现次数，对于同一事件不重复计算。处理不同事件的数目越多，事件覆盖率越高。但事件覆盖率高并不意味着其他覆盖度量效果好。如图 8-17 所示，其状态图要处理 4 种事件：e_1、e_2、e_3、e_4。要达到事件覆盖率 100%，只需遍历转移 T_1、T_2、T_3、T_4，及状态 S_1 和 S_2，便能满足要求。转移 T_5 上的 e_2 被看成是重复事件没有考虑在内，致使转移 T_5 和状态 S_3 没有被覆盖到。按照状态图，对于事件 e_2 的处理有两种情况：一是系统处于状态 S_1 并满足条件 c_2，系统执行动作 a_2；二是系统处于状态 S_3 并满足条件 c_5，系统执行动作 a_5。在如图 8-17 所示的例子中，不考虑第二种情况是可以使事件覆盖覆盖率达到 100%。使用状态-事件覆盖度量，可以避免上述状态与转移的遗漏。为了达到状态-事件覆盖率为 100%，需要涵盖状态-事件组合：(S_1，T_1)、(S_1，T_2)、(S_2，T_4)、(S_2，T_3)、(S_3，T_5)。这样对于图 8-17 中的例子也满足覆盖所有状态与转移条件。

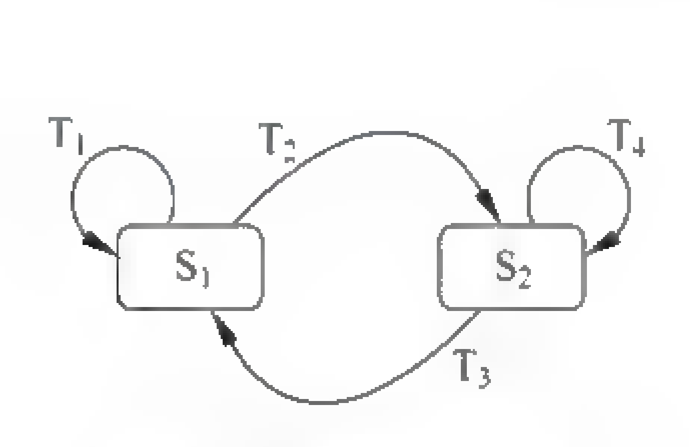


图 8-16　状态转移图

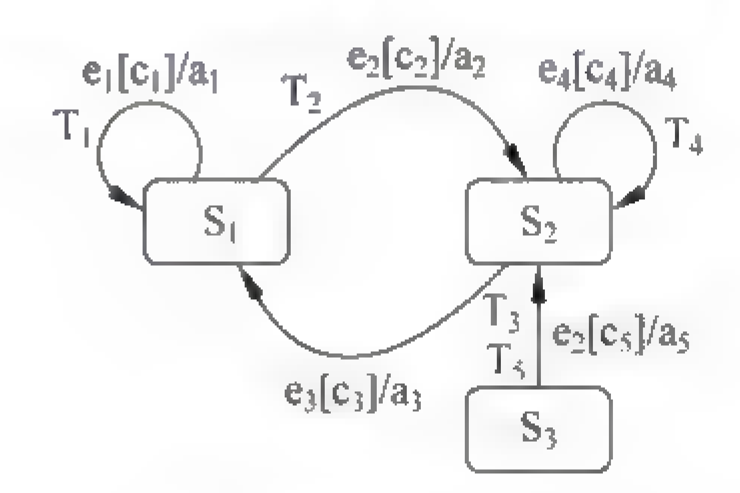


图 8-17　状态转移图

除了上述 4 种覆盖度量外，在基于状态图测试中，还可以考虑其他覆盖度量，如条件覆盖、动作覆盖、事件-条件覆盖等。本节不对这几种覆盖加以讨论，留给读者去分析思考。在实际项目中选择哪种覆盖度量，是根据项目特点，用户需求，及时间与人力预算等方面来决定。如果用户关心所有可能事件是否得到系统响应，可选择事件覆盖；如果考虑到对于同一事件，由于系统出于不同状态需要作出不同响应，则可以选择状态-事件覆盖；如果用户希望了解事件-条件-行动组合是否被执行过，则可以选择转移覆盖。

8.4　总结

基于状态的软件测试是一种基于模型的测试技术，也就是通过建立描述系统行为的状态机，来自动生成测试用例。

模型的质量直接关系到基于状态的软件测试的质量。本章介绍了状态转换图与状态图。状态转换图是一种"平的"结构，描述复杂系统时，很容易产生"状态爆炸"问题。状态图采取层次化结构，可以屏蔽内部复杂性，便于系统的分析。另外，状态图还有其他良好的属性，在实际项目中得到广泛的应用。为了生成测试用例，需要将层次化的状态图转为"平面"状态转换图，通过遍历所得到的有向图，自动生成测试用例。

基于状态的测试技术按照一定的步骤进行：

(1) 创建图形化规格说明书。

(2) 产生中介规格说明书。

(3) 生成测试规格说明书。

基于状态测试的覆盖分析，是根据状态机或状态图相关元素或其组合来构造覆盖率、如状态覆盖率、转移覆盖率和状态-事件覆盖率等。这些覆盖率决定产生测试用例方法和数目，以及测试结束条件。

8.5 参考文献

[1] Mealy, G. H.. A Method for Synthesising Sequential Circuits. *Bell System Technology Journal*. 34(5), September 1955: pp. 1045-1079

[2] Moore, E. F.. Gedanken-Experiments on Sequential Machines. *Annals of Mathematics Studies*. Vol. 34, Automata Studies, Priceton, NJ: Princeton University Press, 1956: pp. 129-153

8.6 思考与练习

1. 试用自己的话，描述什么是基于状态的测试。

2. 比较 Moore 与 Mealy 状态转换图，各自有什么特点？为什么说两种状态机是等价的？

3. Harel 状态图有哪些属性？试比较 Harel 状态图与 UML 状态图。

4. 利用图形化技术进行测试的一般步骤是什么？为什么说基于状态图的测试步骤是其一个实例？

5. 编写一个程序，遍历图 8-13 中所有的路径(从始点 Sb 到终点 Se)并打印出。

6. 什么是测试场景？测试场景与测试用例是什么关系？

7. 基于状态的测试的覆盖分析什么？满足"状态"覆盖意味着满足"转移"覆盖吗？为什么？

8.7 进一步阅读

Larry Apfelbaum, J. Doyle. Model-based testing. Proceedings of the 10th International Software Quality Week (QW 97), 1997

R. V. Binder. Testing Object-oriented System: Models, Patterns, and Tools. Addison-Wesley, 1999

Tsun S. Chow. Testing Design Mmodeled by Finite-state Machines. IEEE Transactions on Software Engineering. 4(3), May 1978: pp. 178～187

D. Harel. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 8(3), 1987: pp. 231～274

Hyoung Seok Hong, Young Gon Kim, Sung Deok Cha. A Test Sequence Selection Method for Statecharts. The Journal of Software Testing. Verification & Reliability, 10(4), December 2000: pp. 203～227

Jeff Offutt, Shaoying Liu, Aynur Abdurazik. Generating Test Data from State-based Specifications. The Journal of Software Testing. Verification and Reliability, 13(1), March 2003: pp. 25～53

Harry Robinson. Finite State Model-based Testing on a Shoestring, San Jose, CA, USA: Proceedings of the 1999 International Conference on Software Testing Analysis and Review (STARWEST 1999), Software Quality Engineering. October 1999

Denning, P. J., J. B. Dennis, J. E. Qualitz. Machines, Languages and Computation. Englewood Cliffs. NJ: Pretice-Hall, 1978

江曼，E天青，潘金贵. 基于 UML 状态图的面向对象软件测试用例生成. 计算机科学，2006

张毅坤，施凤鸣，姚全珠. 基于 UML 的状态图的类测试用例自动生成方法. 计算机工程. 2003

网站：http://www.model-based-testing.org/，和 http://www.geocities.com/model_based_testing/online_papers.htm 有很多有关基于模型的测试资料

第 9 章

面向对象的应用测试

面向对象(Object Oriented,OO)程序设计的基本单元是类和对象。如图 9-1 所示,一个面向对象的应用是由多个类实例集组成,它们通过互相发送消息和调用方法产生联系。一个应用通常是由对象簇(cluslel)所组成,例如:与用户界面相关的若干对象和与数据相关的若干对象组成不同的对象簇。在每一个簇中,对象之间相互发送消息以完成任务。对象之间很少或者根本没有全局数据。

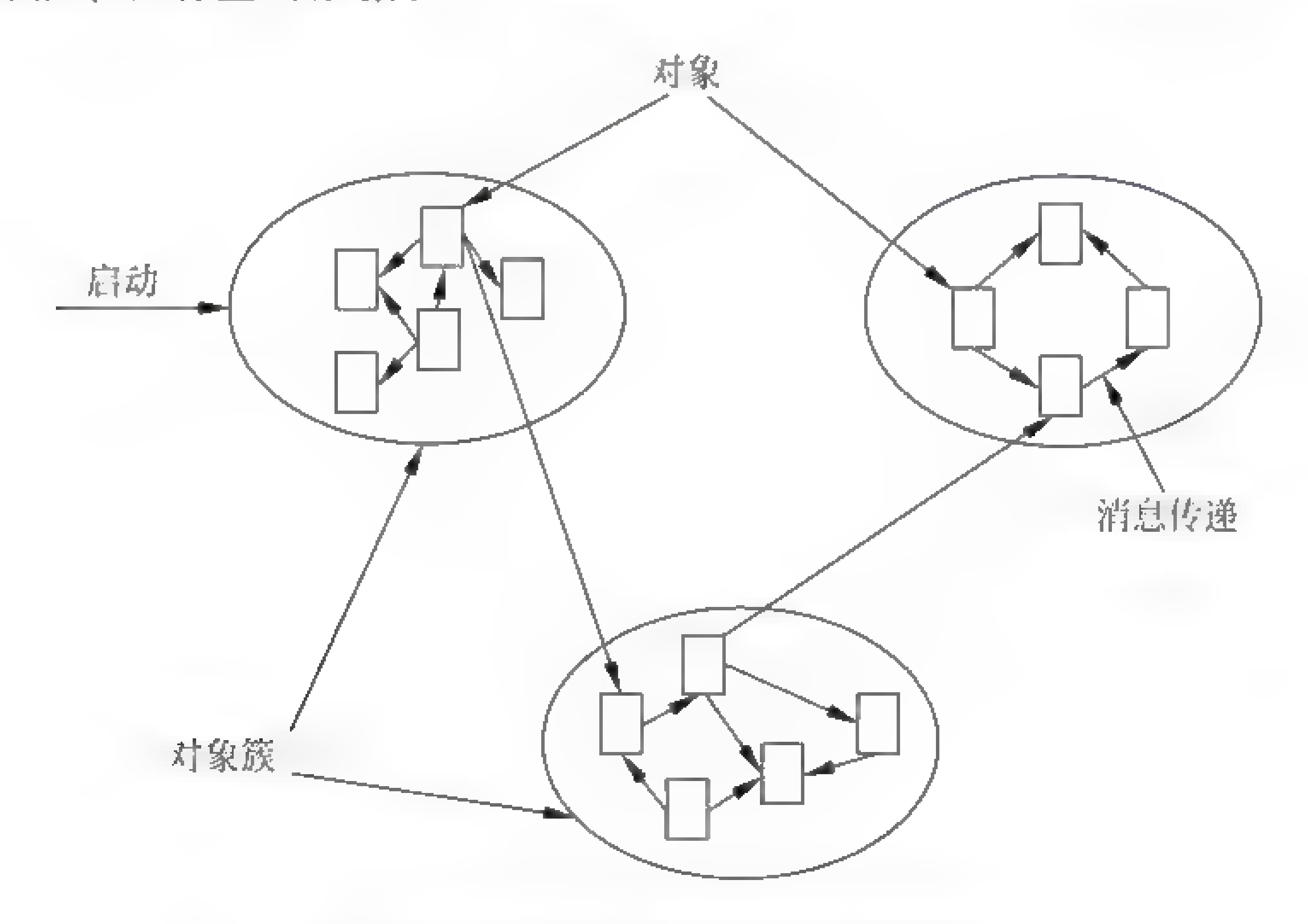


图 9-1 面向对象应用运行时的对象和对象簇

面向对象测试从评估 OOA (Object Oriented Analyze) 和 OOD(Object Oriented Design)模式的正确性和一致性开始。测试策略发生了改变:

(1) 封装扩展了单元的概念。

(2) 集成测试关注于类和它们在一个线程中的执行或在一个用例中的执行。

(3) 确认测试使用传统的黑盒方法。

测试用例设计利用传统方法，但也包括一些特性。

OO测试策略必须考虑OO应用的特点。类是最小的可测试单位。对象拥有状态，测试方法必须考虑这些。类测试和单元测试是等价的，要测试类中的操作和类的状态行为。OO测试关注对象的状态以及它们之间的相互作用。继承为方法定义新的语境。被继承方法的行为可能发生改变而且如果所调用的方法发生了改变，那么所有调用该方法的方法必须被重新测试。

快速阅览：

什么是OO应用测试？ OO应用测试是相关活动的集合，是为了发现存在于OOA、OOD、类、方法（操作）以及类间交互方面的错误。为了完成这些活动，在OO应用要使用包括静态评审和动态执行测试在内的测试策略。

由谁来负责OO应用测试？ OO项目的工程师和其他与项目有关的涉众（管理者、客户和最终用户）都要参与OO应用测试。

为什么OO应用测试如此重要？ 如果最终用户碰到错误并动摇了他们对OO应用的信心，那么这个OO应用就失败了。所以在OO应用投入使用之前，OO工程师必须努力消除尽可能多的错误。

OO应用测试步骤是什么？ 开始是单元测试，集中于测试类及其方法；然后是集成测试，集中于测试类与类间的交互；最后是系统测试。

有哪些工件形成？ 在一些情况下，会生成OO应用测试计划。在每一种情况下，要为OO应用测试生成一组测试用例并将测试结果存档以便将来软件维护所用。

如何确保我们准确地完成了任务？ 尽管永远不能保证你已经执行了所要求的每一个测试，但能肯定测试中已经发现的错误（并且已修正了这些错误）。另外，如果已经制定了一个测试计划，则可以检查以保证所有测试任务已被完成。

9.1 OO测试方法

面向对象软件的体系结构由一系列分层的子系统组成，它们封装了协作的类。每个系统元素（子系统和类）都帮助实现系统需要的功能。必要在各个不同的层次上测试OO系统以便发现在类相互协作时和子系统跨体系结构层次通信时可能发生的错误。

OO测试在策略上和传统的系统测试类似，但是，方法技巧上存在不同。因为OOA和OOD模型在结构和内容上类似于最终的OO程序，所以“测试”从对这些模型的评审开始。一旦代码生成，便开始“小规模”类测试，然后检查类操作和类间协作是否有错误发生。当类被集成并形成子系统，结合基于故障的方法和基于使用的测试技术完全地测试协作类。最后，用例（use-case，作为OOA模型的一部分而开发）被用于发现在软件确认级的错误。

传统的测试用例设计以软件的“输入-处理-输出”形式或单个模块的具体算法为驱动。OO测试集中于合适的操作顺序来检查类的状态。

9.1.1 OO概念对测试用例设计影响

随着分析与设计模型的演进，OO类是测试用例设计的目标。因为属性和操作是被封装的，所以从该类之外测试其操作效率不高。虽然封装是OO的本质概念，但是它可能会成为测试的小障碍，如Binder[1]所说，“测试需要报告一个对象的具体和抽象状态”，然而，除非提供了内置操作来报告类属性的值，否则，难以获得对象的状态快照，即在某时点对象的状态的取值。因此，封装使得这些信息在某种程度上难以获得。

继承给测试用例设计者也带来了额外的挑战。我们知道，即使继承达到了复用目的，对于每个新的使用语境(Context of Usage)，被复用的部分也需要重新测试。此外，多重继承增加了需要测试的语境数量[2]从而使测试进一步得复杂化。如果从超类导出的子类被用于相同的问题域，则有可能把超类导出的测试用例集用于子类的测试。然而，如果子类被用于完全不同的语境，则超类的测试用例将没有多大用处，必须设计新的测试用例集。

9.1.2 传统测试用例设计方法的可用性

第2章描述的白盒测试方法可用于测试类所定义的操作。基本路径、循环测试或数据流技术可以帮助保证测试到操作中的每一条语句，然而，很多类操作的简洁结构导致某些人认为：将用于白盒测试的努力用于类级别的测试可能会更好。

黑盒测试方法对OO系统同样适用，就像其适用于传统软件工程方法所开发的系统。在设计黑盒及基于状态测试时，用例可以提供有用的输入信息。

9.1.3 基于故障的测试

在OO系统中基于故障的测试(Fault based Testing)目标是发现似是而非的故障。因为产品或系统必须符合客户需求，因此，基于故障的测试从分析模型开始。测试员查找可能的故障(即系统实现中可能产生错误的部分)。为了确定是否存在这些故障，需设计测试用例以测试OO设计或OO代码。

考虑一个简单的例子。软件工程师经常在问题的边界处犯错误，例如，当测试SQRT操作(该操作对负数返回错误)时，我们尝试边界：一个靠近零的负数和零本身。“零本身”用于检查是否程序员犯了如下错误：

```
if(x>0) calculate_the_square_root();
```

此错误在于条件没有将“零本身”考虑在内；而正确的 if 语句应是：

```
if(x>= 0) calculate_the_square_root();
```

另一个例子，考虑布尔表达式：

```
if(a && !b || c);
```

多条件测试和相关的技术用于探查在该表达式中某种可能存在的故障，如：

- “&&”应该是“||”。
- “!”是必需的，但被省去。
- “!b || c”应该放在括号( )中。

对于每个可能的故障，可设计测试用例，使得不正确的表达式运算失败。在上面的表达式中，(a=0，b=0，c=0)将使得所给的表达式值为“假”，如果“&&”本应该为“||”，则该代码做了错误的事情，有可能分叉到错误的路径上。

当然，这些技术的有效性依赖于测试员如何感觉什么是“似乎可能的故障”。如果 OO 系统中的真实故障被感觉为“不像真实的”，则实质上本方法不比任何随机测试技术更好。然而，如果分析和设计模型可以帮助深入洞察什么地方可能出错，则基于故障的测试可以以相当少的工作量来发现大量的错误。

集成测试(OO 语境下)在操作调用或消息连接中查找可能的故障，在此语境下，会遇到 3 种类型的故障：不期望的结果、使用了错误的操作/消息、不正确的调用。为了在函数(操作)调用时确定可能的故障，必须检查操作的行为。

集成测试既应用于属性又应用于操作。对象的“行为”通过其属性被赋予的值来定义。测试应该检查属性以确定是否对对象行为的不同类型产生合适的值。

应该注意，集成测试试图发现使用服务的对象或客户对象中的错误，而不是提供服务的对象中的错误。用传统的术语来说，集成测试的关注点是确定调用代码中是否存在错误，而不是被调用代码中是否存在错误。调用操作作为线索来发现测试需求，以便检查调用代码。

9.1.4 OO 编程对测试的影响

OO 编程可通过多种方式对测试产生影响。根据面向对象编程的方法，

- 某些新类型的错误出现了。
- 某些类型的错误变得不大可能(不值得去测试)。
- 某些类型的错误变得更有可能(值得现在测试)。

调用一个操作时，也许很难分辨到底执行了什么代码，也即这个操作可能属于许多类中的一个。而且也很难确定一个参数的确切类型或类。当代码存取它时，可能得到意想不到的值。可通过一个传统的函数调用例子来理解这点：

x = func(y);

对于传统的软件测试，测试者仅需考虑属于 func 的所有行为。在 OO 软件测试过程中，测试者必须考虑 Base::func()、Derived::func()的行为，等等。每次 func 被调用时，测试者必须考虑所有不同行为的联合。如果遵循好的 OOD 实践并且超类与子类(C++术语中分别称为基类与派生类)之间的区别是有限的，这会变得容易些。基类与派生类的测试方法本质上是一样的。

测试面向对象中的类操作类似于测试需要一个函数参数的代码，并调用它。继承是一种产生多态操作的便捷方式。在调用处，重要的不是继承而是多态。继承的确使搜寻测试需求变得更加直接(即有继承便要考虑测试)。

是否因面向对象软件的架构和构造而使某些类型的错误对于一个面向对象的系统变得更有可能，而对于其他系统变得不大可能？答案是肯定的。例如，由于 OO 中的操作通常很小(功能、大小等方面)，更多的时间将花在集成上，因为集成错误出现的几率更大。

9.1.5 测试用例和类层次

正如本章前面指出的，继承并不排除对所有派生类进行彻底测试的需要。事实上，它其实可能将测试过程复杂化。考虑以下情况：类 Base 包含操作 inherited 和 redefined。类 Derived 重定义 redefined 以服务于一个局部语境(Context)。毫无疑问，Derived::redefined()必须被测试，因为它代表了一个新的设计和新的代码。但 Derived::inherited()必须被重测吗？

如果 Derived::inherited()调用 redefined 并且 redefined 的行为已改变，那么 Derived::inherited()可能错误地处理这个新行为。所以，即使设计和代码没有改变也需要新的测试。需要注意的是，仅需针对 Derived::inherited()的所有测试的一个子集进行。如果 inherited 的部分设计和代码不依赖于 redefined(即其既不调用它也没有任何代码间接调用它)，则该代码在派生类中不必重测。

Base::redefined()和 Derived::redefined()是两个具有不同规格和实现的不同操作。每个都有一套派生自规格和实现的测试需求。那些测试需求探究可能的错误：集成错误、条件错误、边界错误等，但操作可能很相似。它们的测试需求会重叠。面向对象设计得越好，重叠越大。新的测试仅需生成那些未被 Base::redefined()的测试所满足的 Derived::redefined()的需求。

总的来说，Base::redefined()的测试适用于类 Derived 的对象。测试输入可能对基类和派生类都合适，但期望的结果可能不同于派生类。

类层次的设计表示提供了对继承结构的深入洞察，继承结构被用在基于故障的测试中。考虑如下情形：一个命名为 caller 的操作只有一个参数，并且该参数是某基类的引用。当 caller 被传递给该基类的派生类时将发生什么事情？可能影响 caller 的行为差异是什么？对这些问题的回答可能导向特殊测试的设计。

9.1.6 基于场景的测试

基于故障的测试遗漏了两种主要错误：不正确的规格和子系统间不正确的交互。任何一种错误都会对软件质量（需求的一致性）造成损害。第一种错误发生时，产品没有做客户想要的。它可能做了错误的事也可能忽略了重要的功能。与子系统交互相关的错误发生在一个子系统的行为创建某状况（例如事件、数据流）时，该状况会使另一个子系统失效。

基于场景的测试（Scenario-based Testing）集中于用户做什么，而不是产品做什么。这意味着要捕获用户必须完成的任务（通过用例），然后运用它们及其变体进行测试。

场景揭露交互错误。要做到这一点，测试用例必须比基于故障的测试用例更复杂更现实。基于场景的测试倾向于在单一测试中测试多个子系统（用户并不限制自己在一个时间使用一个子系统）。

例如：一个文本编辑器的基于场景的测试设计。下面是一个非正式的用例：

用例：修订最后草案

背景：打印"最后"草案，阅读它，发现一些在屏幕图像上看来并不明显的恼人的错误。这个用例描述了当其发生时出现的事件序列。

1. 打印整篇文档。
2. 在文档中来回浏览，对某些页面做些修改。
3. 每当一页被改变了，就把它（被修改的页）打印。
4. 有时打印一系列连续页面。

该场景描述了两件事：一个测试和特定的用户需要。用户的需要是显然的：

(1) 打印单个页面的方法。

(2) 打印区间页面的方法。

就测试而言，需要对在打印后的编辑。（也包括相反的操作）进行测试。测试者希望发现因打印功能造成的编辑功能中的错误，即这两个功能不是完全独立的。

用例：打印一份新的副本

背景：有人向用户要一份文档的全新副本。它必须被打印出来。

1. 打开文档。
2. 打印它。
3. 关闭文档。

同样，测试方法相对明显。文档是以前的任务产生的，会影响到这次的打印吗？在许多现代编辑器中，文档会记住上次是怎样被打印的。默认地，它们下次会以相同的方式打印。在"修订最后草案"场景之后，只是选择菜单中的"打印"命令并单击对话框中的"打印"按钮会造成最后被修改的页面再次打印。因此，依照编辑器，正确的场景看上去像这样：

用例：打印一份新的副本

1. 打开文档。
2. 选择菜单中的“打印”命令。
3. 检查是否在打印一个页面区间；如果是，则单击“打印整个页面”选项。
4. 单击“打印”按钮。
5. 关闭文档。

但这个场景一个潜在的规格错误。编辑器没有做用户期望它要做的事。客户经常忽视上面第 3 步中的检查。当他们小跑到打印机前发现是 1 页而不是他们想要的 100 页时会很恼火。客户的不满意预示着规格缺陷。

一个测试用例设计师可能遗漏测试设计中的这种依赖性，但该问题很可能在测试期间浮现。

9.1.7　测试表层结构和深层结构

面向对象编程可能有两种效果：改变了产品的深层结构，也可能影响了用户所看到的表层结构。表层结构指从外部可观察的结构，即最终用户立即可见的结构。很多 OO 系统的用户不是在执行功能，而是被给定一些对象，以特定方式来操纵这些对象。但是不管接口是什么，测试仍然基于用户任务进行。捕获这些任务涉及理解、观察以及与代表性用户（以及很多有价值的非代表性用户）的交谈。

在这方面，传统方法和 OO 方法在细节上存在某些差异。例如，在传统的具有面向命令的界面系统中，用户可能使用所有命令的列表作为检查表。如果不存在测试场景去执行某命令，测试可能忽略某些用户任务（或界面上有无用命令）。在基于对象的界面中，测试员可能使用所有的对象列表作为检查表。

当设计者以一种新的或非传统的方式来看待系统时，则可以得到最好的测试策略。例如，如果系统或产品具有基于命令的界面，则当测试用例设计者假设操作是独立于对象的，将可以得到更彻底的测试。提出如下问题：“当使用打印机工作时，用户有可能希望使用该操作（它仅应用于扫描仪对象）吗？”不管界面风格怎样，检查表层结构的测试用例设计应该同时使用对象和操作，以找出导致忽视任务的线索。

基于表层结构测试设计可能会遗漏一些东西，比如可能注意不到用户的某些任务；应该测试的重要情况没有被测试到；没有查明特殊子系统之间的交互。例如，代码 A 的一段依赖于代码 B 的一段，但似乎没有测试来执行 A 使用 B，这就是一个线索。我们没有注意到用户的一个任务。着眼于深层结构则能揭露这些疏忽。

深层结构指 OO 程序的内部技术细节，即通过检查设计和代码来理解结构。深层结构测试用以测试依赖、行为和通信机制，这些是建立 OO 软件设计模型的一部分。

分析和设计模型被用作深层结构测试的基础。例如，UML 协作图或部署模型描述了对象和子系统间的协作，这种协作从外部看不可见。那么测试用例设计者会问：“我们已经

捕获了某些测试任务，来测试在协作图中描述的协作吗？如果没有，为什么？”

9.2 在类级别上可用的测试方法

第1章曾提到软件测试从“小型”测试开始，慢慢进展到“大型”测试。对OO系统的小型测试着重于单个类和类封装的操作。在OO测试中，可以用随机测试和划分测试来测试类[3]。

9.2.1 面向对象的随机测试

为了简要说明这些测试方法，考虑一个银行应用，其中Account类有下列操作：open()、setup()、deposit()、withdraw()、balance()、summarize()、creditLimit()和close()[3]，尽管每一个操作均可应用于Account，但是领域问题的本质给它们增加了一些限制(如，在应用其他操作前必须先打开账户，在所有操作完成后才关闭账户)。即使有了这些限制，也存在操作的很多排列。对于一个Account实例，最小的行为生命历史包括下面操作，用正则表达式表示如下：

open • setup • deposit • withdraw • close

这表示了对Account的最小测试序列，其中符号“•”表示操作顺序地执行。然而，更为一般的序列可以表述如下：

open • setup • deposit • [deposit | withdraw | balance | summarize | creditLimit] * • withdraw • close

符号“|”表示可供选择的操作，符号“*”表示可以出现零次或多次。基于上述一般的序列和根据正则表达式规则，用随机方法可以产生一系列不同的操作序列，每个序列对应于一个测试用例，例如测试用例r1对应的序列如下：

open • setup • deposit • deposit • balance • summarize • withdraw • close

测试用例r2对应的序列如下：

open • setup • deposit • withdraw • deposit • balance • creditLimit • withdraw • close

执行这些和其他的随机顺序测试以测试不同的类实例生命周期。

9.2.2 在类级别上的划分测试

与传统软件的等价划分相似，采用划分测试(Partition Testing)可以减少测试类所需的测试用例的数量。对输入和输出进行分类，设计测试用例以处理每个类别。但是，如何导出

划分类别呢?

基于状态的划分是根据类操作改变类的状态的能力来划分类操作的。再来考虑 Account 类，状态操作包括 deposit 和 withdraw，而非状态操作包括 balance、summarize 和 creditLimit。分别独立地测试改变状态的操作和不改变状态的操作，因此，对于测试用例 p1 有：

open • setup • deposit • deposit • withdraw • withdraw • close

对于测试用例 p2 有：

open • setup • deposit • summarize • creditLimit • withdraw • close

测试用例 p1 测试改变状态的操作，而测试用例 p2 测试不改变状态的操作(除了那些在最小序列中的操作)。

基于属性的划分是根据操作使用的属性来划分类的操作。对 Account 类，用属性 creditLimit 来定义划分，操作被分为 3 个类别：使用 creditLimit 的操作、修改 creditLimit 的操作、不使用或不修改 creditLimit 的操作。然后对每个划分设计测试序列。

基于类别的划分是根据各自完成的一般功能来划分类的操作。例如，在 Account 类中的操作可分为初始化操作(open、setup)、计算操作(deposit、withdraw)、查询操作(balance、summarize、creditLimit)和终止操作(close)。

9.3 类间测试用例设计

OO 系统开始集成后，测试用例的设计变得更复杂。正是在此阶段，必须开始对类间的协作测试。为了说明类间测试用例生成[3]，扩展在 9.2 节中引入的银行例子，使包含如图 9-2 所示的类和协作，图中箭头的方向指明消息的传递方向，标注则指明调用的操作，是消息中蕴涵的一系列协作。

与个体类的测试一样，类协作测试可通过应用随机和划分方法、基于场景的测试和行为测试来完成。

9.3.1 多个类测试

Kirani 和 Tsai[3] 建议采用下面的步骤生成多个类测试用例：

(1) 对每个客户类，使用类操作列表来生成一系列随机测试序列。操作将发送消息给其他服务器对象。

(2) 对所生成的每个消息，确定协作者类和对应的服务器(协作者)对象中的操作。

(3) 对服务器对象(已经被来自客户对象的消息调用)中的每个操作确定传递的消息。

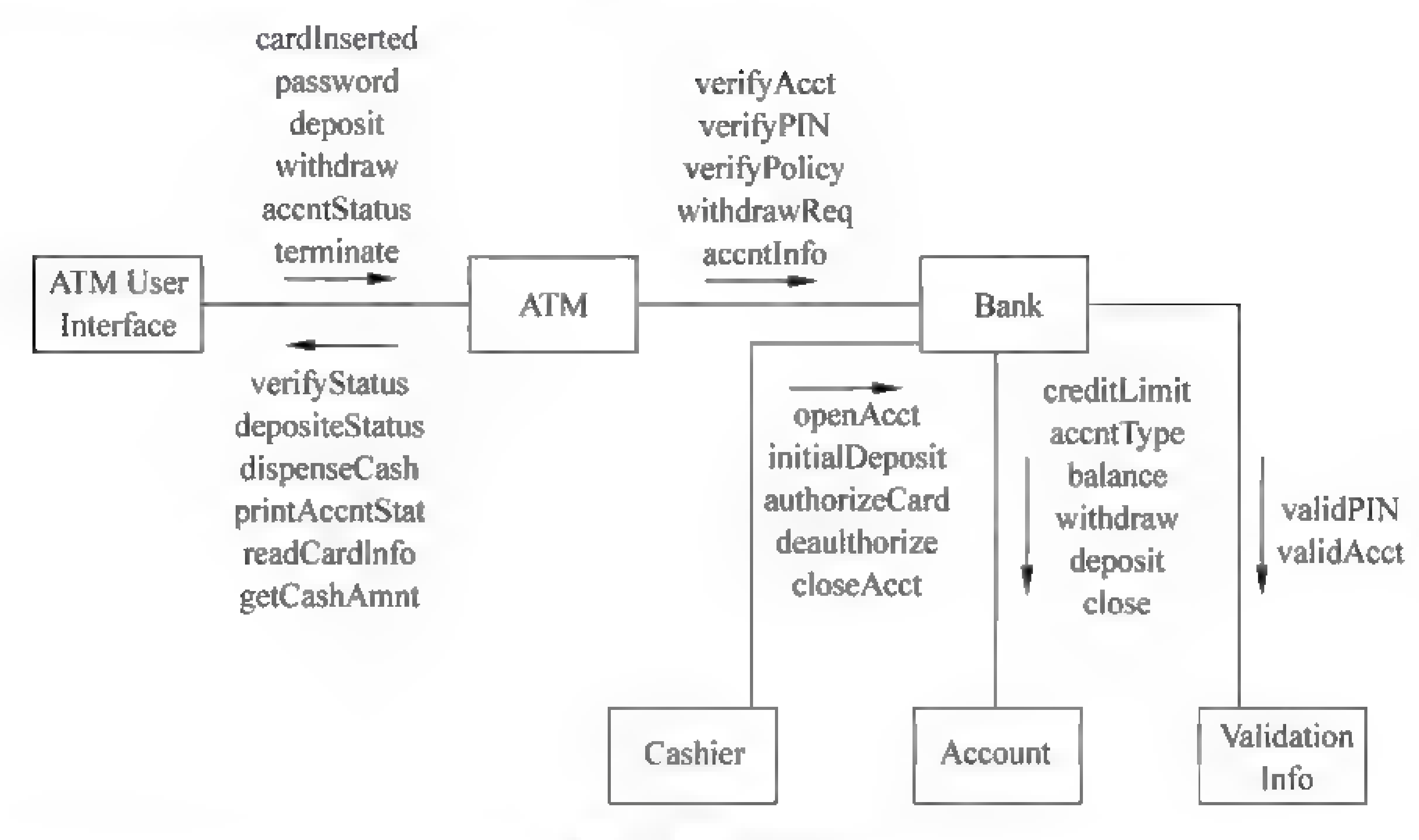


图 9-2 类及其协作

(4) 对每个消息，确定下一层被调用的操作并把这些操作放入到测试序列。

为了说明[3]，考虑 Bank 类相对于 ATM 类的(见图 9-2)的操作序列：

verifyAcct • verifyPIN • [[verifyPolicy • withdrawReq] | depositReq | acctInfoREQ]^n

对 Bank 类的随机测试用例可能是测试用例 r3：

verifyAcct • verifyPIN • depositReq

为了考虑该测试中所涉及到的协作者，要考虑测试用例 r3 中提到的与每个操作相关联的消息。Bank 必须和 ValidationInfo 协作以执行 verifyAcct()和 verifyPIN()，Bank 必须和 Account 协作以执行 depositReq()，因此，对于协作，新的测试用例是测试用例 r4：

verifyAcctBank • [validAcct ValidationInfo] • verifyPINBank •
[validPINValidationInfo] • depositReq • [depositaccount]

多个类划分测试的方法类似于单个类划分测试的方法。单个类划分就像在 9.2 节所讨论的那样。然而，扩展测试序列以包括那些通过发送消息给协作类而被激活的操作。另一种方法是基于特殊类的接口来划分测试，如图 9-2 所示，Bank 类接收来自 ATM 和 Cashier 类的消息，因此，可以通过将 Bank 中的方法划分为服务于 ATM 和服务于 Cashier 的操作来测试。基于状态的划分(9.2 节)可用于进一步精化划分。

9.3.2 从行为模型导出的测试

第 8 章讨论了使用状态图表示类动态行为的模型。类的状态图可用于导出测试序列，来测试类(以及那些与其协作的类)的动态行为。图 9-3[3]给出了前面讨论的 Account 类的

状态图。根据该图，初始变迁是从 Empty acct 到 Setup acct 状态，类的实例的大多数行为发生在 Working acct 状态，最终的 Withdrawal 和 Close 事件使得 Account 类分别向 Nonworking acct 和 Dead acct 状态变迁。

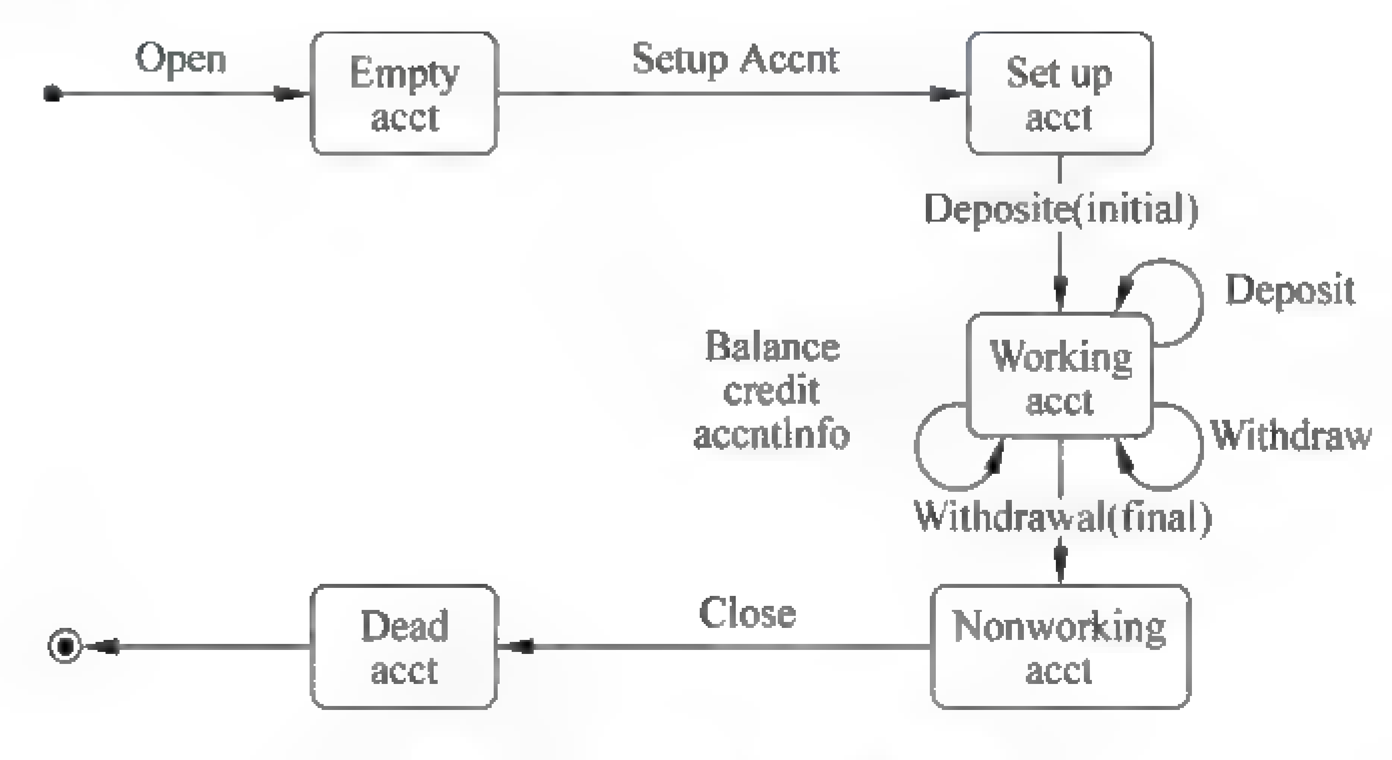


图 9-3　Account 类的状态图

Account 类的一般的序列可以表述如下(注意：该序列类似于 9.1 节讨论的一般序列)：

open • setup • deposit • [deposit | withdraw | balance] * • withdraw • close

设计的测试应该涵盖所有的状态[3]，即，操作序列应该?? 使 Account 类产生所有可能的状态。对于测试用例 s1：

open • setupAccnt • deposit(initial) • withdraw(final) • close

注意：该序列等同于 9.1 节讨论的最小测试序列。加入其他测试序列到最小序列中得到测试用例 s2：

open • setupAccnt • deposit(initial) • deposit • balance • credit •
withdraw(final) • close

测试用例 s3：

open • setupAccnt • deposit(initial) • deposit • withdraw • accntInfo •
withdraw(final) • close

根据 Account 类的一般序列，仍然可以导出更多的测试用例以保证已经适当地测试了类的所有行为。当类行为表现为与一个或多个类协作的时，使用多个状态图跟踪系统的行为流。

可以按“宽度优先的方式”遍历状态模型[4]，在测试语境下，宽度优先指的是：用一个测试用例测试单个变迁，测试新的变迁时，仅使用以前被测试过的变迁。

银行系统中的 CreditCard 对象，如图 9-4 所示。CreditCard 的初始状态是 undefined (即，没有提供信用卡号)。通过在销售中读信用卡，对象进入 defined 状态，即定义属性

card number 和 expiration date 以及银行特定的标识符。当发送请求授权时，信用卡被提交(submitted)；当接收到授权时，信用卡被核准(approved)。CreditCard 从一个状态到另一个状态的变迁可以通过导出引致变迁发生的测试用例来测试。测试类型宽度优先的方法使得不会在测试 undefined 和 defined 之前测试 submitted，这样将使用以前尚未测试的变迁，因此会违反宽度优先准则。

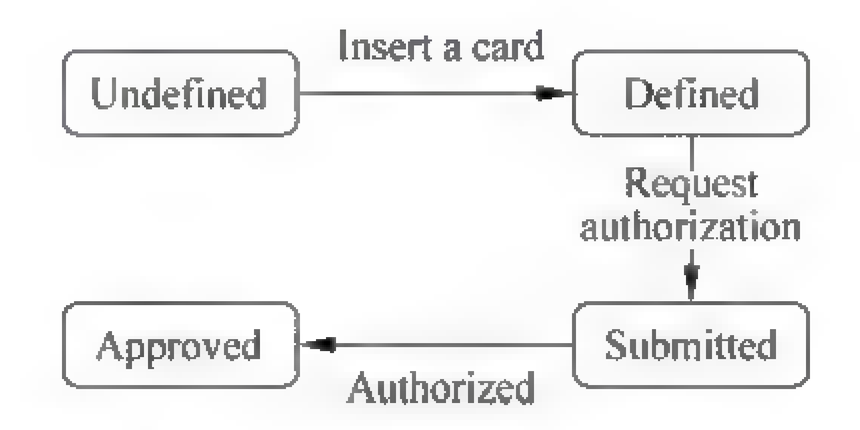


图 9-4 CreditCard 对象状态转移

9.4 总结

尽管面向对象测试的总体目标与传统的软件测试目标一致，即用最小的努力发现最多的错误，但是 OO 测试在策略和方法有所不同。测试的视角被拓宽，从而包括了对分析与设计的评审。另外，测试的重点从过程组件(模块)转向类。对类测试的设计可以使用多种方法：基于缺陷测试、随机测试和划分测试。每种方法都要测试类所封装的操作。设计测试顺序以保证相关的操作都得到测试。类的状态由其属性的值表示，测试其状态以确定是否存在错误。

集成测试可以用基于使用的策略来完成。基于使用的测试构造具有层次系统，始于不调用服务器的类。集成测试的用例设计方法可以使用随机与划分测试技术。另外，基于场景的测试和由行为模型生成的测试可用于测试类及其协作者。一个测试序列追溯类协作的操作流。

9.5 参考文献

[1] Binder, R. V.. *Testing Object-Oriented System: A Status Report*, *American Programmer*. vol. 7, no. 4, April 1994, pp. 23～28

[2] Binder, R. V.. *Testing Object-oriented System: Models, Patterns, and Tools*. Addison-Wesley, 1999

[3] Kirani, S., W. T. Tsai. *Specification and Verification of Object-Oriented Programs*. Technical Report TR 94-64, Computer Science Department, University of Minnesota, December 1994

[4] McGregor, J. D., T. D. Korson. Integrated object-oriented testing and development processes. *Communications of the ACM*. vol. 37, no9, September 1994, pp. 59～77

9.6 思考与练习

1. 试用自己的话，描述类是OO系统中最小的可接受的测试单元。

2. 如果现存的类已经被彻底测试过，为什么我们还需要重新测试由那个现存的类实例化的子类？我们能使用为现存的类所设计的测试用例吗？

3. OO集成测试的特点？

4. OO测试用例的设计方法有哪些？

5. 什么是基于缺陷的测试？什么是基于场景的测试？它们的异同是什么？

6. 类级别上的测试意义是什么？有什么方法可以利用？

7. 类之间的测试意义是什么？有什么方法可以利用？

8. 测试类的操作时，因为什么原因造成测试的困难？

9. 深层测试是测试OO应用的什么方面？

10. 类的行为的模型可以基于什么分析模型？

9.7 进一步阅读

R. V. Binder. Testing Object-oriented System: Models, Patterns, and Tools. Addison-Wesley, 1999

David C. Kung, Pei Hsia, Jerry Gao. Testing Object-Oriented Software. Wiley-IEEE, 1998

J. D. McGregor & D. A. Sykes. A Practical Guide to Testing Object-Oriented Software. Addison-Wesley, 2001

John D. McGregor & David A. Sykes. Object-Oriented Software Development: Engineering Software for Reuse, International Thompson Publishing, 1991

网站：http://oo-testing.com/bib/

Kaner, C., Pattern: Scenario Testing (draft), http://www.testing.com/test-patterns/patterns/pattern-scenario-testing-kaner.html

第10章

Web应用软件测试技术*

Web工程项目开发过程中似乎自始至终都充满着紧张的气氛。当进行问题陈述(Formulation)、计划、分析、设计和构建时，利益攸关者由于关心来自其他Web应用的竞争和受制于客户的要求，担心将会错过一个市场机会，所以利益攸关者一直给Web应用施加压力，使其尽快投入使用。这样做的后果是，在Web工程项目开发中经常处于后续阶段的技术活动，比如Web应用测试，时常会被忽视，这可能是一个灾难性的错误。为了避免这种错误发生，Web工程项目组必须确保Web工程项目的每一工件都具备良好的质量。华莱士(Wallace)和他的同事们[1]这样陈述：

> “不要等到项目结束时才进行测试，在写第一行代码之前就开始进行测试。持续有效的测试会使你开发一个更持久耐用的网站。”

因为不能对分析与设计进行传统意义上的测试，所以Web工程项目组除了进行可执行的测试外，还应该实施规范的技术审查，目的是使最终用户使用Web应用之前发现并修改错误。

快速阅览：

什么是Web应用测试？ Web应用测试是相关活动的集合，目的是为了发现存在于Web应用中的内容、功能、易用性、导航、性能、容量、安全性等方面的错误。为了完成这些活动，在Web工程中要使用包括静态评审和动态执行测试在内的测试策略。

由谁负责Web应用测试？ Web项目的工程师和其他与项目有关的利益攸关者(管理者、客户和最终用户)都要参与Web应用测试。

为什么Web应用测试如此重要？ 如果最终用户碰到错误并动摇他们对Web应用的信心，他们就要去其他地方寻找所需要的内容和功能，这样这个Web应用就失败了。所以在Web应用投入使用之前，Web工程师必须努力消除尽可能多的错误。

---

* 本章基于 Pressman, Software Engineering: A Practitioner's Approach, (6th edition)，第19章。

Web应用测试步骤是什么? Web应用测试开始集中于用户可见的Web应用方面,然后是测试技术与基础设施。执行7个测试步骤:内容测试、界面测试、导航测试、组件测试、配置测试、性能测试和安全性测试。

有哪些工件形成? 在一些情况下,会生成Web应用测试计划。在每一种情况下,要为每个测试步骤生成一组测试用例并将测试结果存档以便将来软件维护所用。

如何确保我们准确地完成了任务? 尽管永远不能保证你已经执行了所要求的每一个测试,但能肯定测试已经发现了错误(并且已修正了这些错误)。另外,如果已经制定了一个测试计划,则可以检查以保证所有计划测试已被完成。

10.1 Web应用测试概念

测试是为了发现软件的错误(并最终修正错误)而运行软件的过程。对Web应用来说,这些最基本的原则不会变。事实上,因为基于Web的系统和应用存在于网络上,并且和很多不同的操作系统、浏览器(或其他界面设备如PDA、手机等)、硬件平台、通信协议、后台(Backroom)应用进行交互,所以对于Web工程师来说寻找错误意味着很重大的挑战。

为了理解在Web工程中测试的目标,必须考虑Web应用质量的多个纬度。在这个讨论中,要特别注意与测试Web工程项目有关的质量纬度,也会考虑测试所发现的错误的本质以及用来发现这些错误的测试策略。

10.1.1 质量的纬度

良好设计体现在Web应用所带来的质量。Web应用质量的评估是通过技术评审和测试。应用一系列的技术评审去评估设计模型的各个组成部分,并应用本章讨论的测试过程去评价Web应用实现。评审和测试会检查下面一个或多个质量纬度[2]:

- 内容(Content)是在句法和语义级别的评估。在句法级,要评估基于文本的文档中的拼写、标点、语法等;在语义级,要评估信息表现的正确性、整个内容对象和相关对象的一致性、无二义性。
- 功能(Function)测试是要发现与客户需求不符的错误。每个Web应用功能要评估其正确性、不稳定性(Instability)、与实现标准的符合性(比如,Java或者XML语言标准)。
- 结构(Structure)评估是要确保正确发布了Web应用的内容和功能,并且是可扩展的,要支持新内容和新功能的增加。
- 易用性(Usability)测试是要确保对每一类用户都要有相应的界面支持,用户要能学习和应用所有需要的导航句法和语义。

- 导航(Navigability)测试要确保所有的导航句法和语义都被测试,从而发现有关导航的任何错误(比如死链接、不合适的链接、错误的链接)。
- 性能(Performance)测试确保在各种操作条件、配置、负载变化的情况下,系统在响应用户交互和处理极限负载时,性能没有出现不可接受的退化。
- 兼容性(Compatibility)测试通过在服务器和客户端不同的配置情况下,执行Web应用。目的是发现跟某种配置相关联的特殊错误。
- 互操作性(Interoperability)测试是确保Web应用能正确地与其他应用或数据库进行交互。
- 安全(Security)测试是评估潜在的易受攻击的弱点并尽量发现这些弱点。任何成功的渗透企图意味着安全性上的漏洞。

Web应用的测试策略和方法用来测试这些质量纬度,在本章的后面部分要讨论这些策略。

> 创新对于软件测试人员来说是件苦乐参半的差事。正像我们知道如何测试一种特殊的技术一样,一种新的软件(Web应用)出现,所有的方法都不灵了。
>
> James Bach

10.1.2 Web应用环境中的错误

对于任何软件,测试的基本目的都是发现错误并修正错误。成功的Web应用测试所发现的错误具有下列一些特征[3]:

(1) 因为对很多种Web应用测试而言,首先是在客户端发现问题出现的证据(比如,通过在一个特定浏览器或PDA或手机上实现的接口),所以Web工程师看到的是一个错误的症状,不是错误本身。

(2) 因为Web应用是在许多不同的配置和不同的环境中实现的,所以发生在某个Web环境中的错误可能很困难或者不可能在该环境之外重现。

(3) 尽管一些错误是由不正确的设计或不合适的HTML(或其他程序语言)代码造成的,但很多错误都能被追踪到Web应用配置。

(4) 因为Web应用存在于客户端/服务器(C/S)体系结构中,所以很难跨越客户端、服务器、网络3个结构层追踪错误。

(5) 一些错误发生在静态操作环境(也就是执行测试的特定配置)中,而其他一些错误发生在动态操作环境中(也就是实时资源负载或与时间有关的错误)。

上述5种错误特征说明在Web工程中,环境在诊断所发现的错误中扮演了一个很重要的角色。在一些情况下(比如内容测试),错误出现的位置很明显,但是在很多其他类型的Web应用测试中(比如导航测试、性能测试、安全测试),错误发生的潜在原因可能很不容易确定。

10.1.3　测试策略

Web 应用测试策略采用对所有软件测试都适用的基本原则，并应用在面向对象系统中所使用的策略。下面的步骤概述了该方法：

(1) 评审 Web 应用的内容模型以发现错误。

(2) 评审 Web 应用的接口模型以保证所有的用例都被考虑到了。

(3) 评审 Web 应用的设计模型以发现导航错误。

(4) 测试用户界面以发现在表现和导航机制中的错误。

(5) 选择功能模块进行单元测试。

(6) 测试贯穿整个体系结构的导航路径。

(7) 在各种不同的环境配置下实现 Web 应用，并测试每种配置的兼容性。

(8) 在 Web 应用及其环境中通过查找易受攻击的漏洞来进行安全性测试。

(9) 进行性能测试。

(10) 由一定数量的最终用户测试 Web 应用，他们和系统的交互结果用来评估内容和导航错误、易用性考虑、兼容性考虑、可靠性和 Web 应用的性能。

因为许多 Web 应用是不断改进的，所以 Web 应用测试也是一个由 Web 支持人员利用已开发的测试用例进行回归测试的持续的过程。

10.1.4　测试计划

计划(Planning)一词对一些 Web 开发人员来说就是魔咒，这些开发人员开始只是希望能够开发出杀手级的 Web 应用，而 Web 工程师认识到计划可以制定一个后续所有工作的路线图，这是值得付出代价的。

Splaine 和 Jaskiel[4] 在他们关于 Web 应用测试的书中写道：

> 除了那些最简单的 Web 站点，都需要某种形式的测试计划。经常会有这样的情况，起初用非规范的测试方法发现很多错误，但这些错误并不都是在第一次发现后就改正过来。这就增加了测试人员的负担。为了进行有效的测试并保证已知的错误已经被改正而没有引入新的错误，他们不仅要设计新的测试，还必须记住前面的测试是如何执行的。

对 Web 工程师来说，问题是如何想出新的测试，这些测试应该注重些什么等。这些问题的答案就包含的测试计划中。

Web 应用的测试计划应该确定：

(1) 测试开始时的任务集。

(2) 每个测试任务完成后产生的工件(Work Product)。

(3) 测试结果被记录、评估和在回归测试时重用的方式。有时候测试计划放在项目计划中,有时测试计划是单独的文档。

Web 应用测试的任务集可描述如下:

(1) 评审利益攸关者的需求,识别用户的关键目的与目标,评审每一类用户的用例。

(2) 制定优先级确保每个用户的目的与目标都被充分测试。

(3) 通过描述将要被执行的测试类型定义 Web 应用测试策略。

(4) 制定一个测试计划:

- 定义一个测试进度表和对每一个测试分配职责。
- 指定自动测试工具。
- 为每一类测试定义用户接受的标准。
- 指定缺陷追踪机制。
- 定义问题报告机制。

(5) 执行"单元"测试:

- 评审内容以发现句法和语义错误。
- 评审内容以保证适度的清晰和许可。
- 进行界面测试以保证正确地操作。
- 测试每一个组件以确保正确的功能。

(6) 执行"集成"测试:

- 根据用例测试界面的语义。
- 执行导航测试。

(7) 执行配置测试:

- 评估客户端的配置和兼容性。
- 评估服务器端的配置。

(8) 执行性能测试。

(9) 执行安全测试。

10.2 测试过程概述

对于 Web 工程的测试过程是从测试最终用户立即可见的内容和界面功能开始的。随后,体系结构设计和导航方面要被测试,用户可能理解也可能不理解这些 Web 应用元素。最后,测试的重点转移到测试对最终用户来说不总是显而易见的技术能力上,如 Web 基础设施和安装、实现问题。

> “通常来说，用于其他应用的软件测试技术，同样能够用于 Web 应用测试……两者不同的是在 Web 环境中技术变化的因素成倍增加了”。
>
> Hung Nguyon[3]

图 10 1 把 Web “金字塔”型的设计过程与测试过程并列。测试流从左到右、从上到下进行，对用户可见的 Web 应用设计部分(“金字塔”塔顶的部分)首先被测试，随后测试基础设施设计部分。要进行 7 个主要的步骤：内容测试、界面测试、导航测试、组件测试、配置测试、性能测试和安全测试。

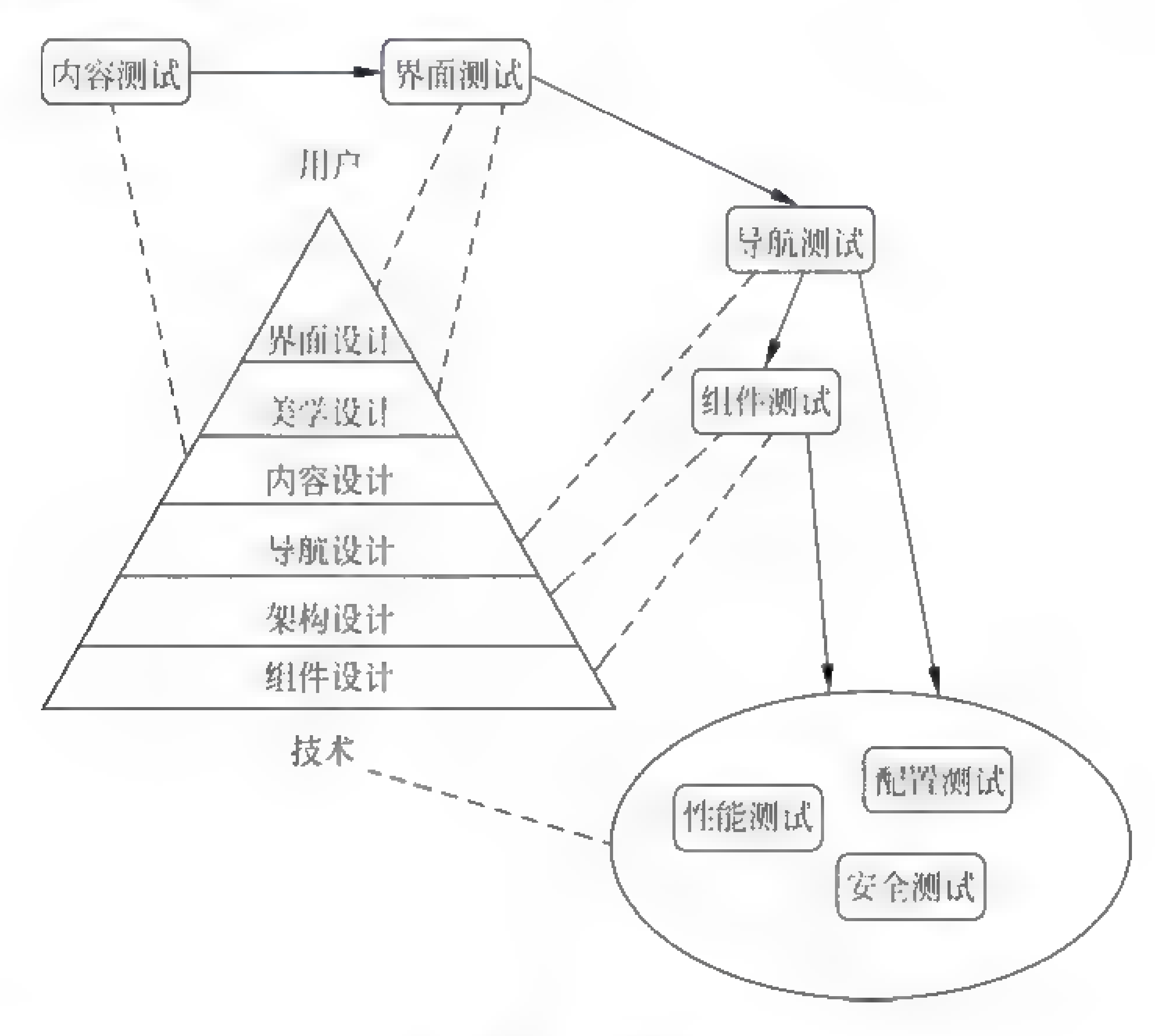


图 10-1 测试过程

内容测试(和评审)要发现内容中的错误，这与在写文档时的审稿有很多相似之处。事实上，-个大的网站可能提供专门的审稿器(Copy Editor)服务来发现排版错误、语法错误、内容不 -致的错误、图形表现错误、交叉引用错误。除了检测静态的内容错误，内容测试也检查来自 Web 应用中数据库系统管理的数据而产生的动态内容。

界面测试是测试交互机制和验证用户界面的美观性。目的是发现由于疏忽而把笨拙的交互、冗长、不一致、二义性引入到界面而引起的错误。

导航测试使用从分析阶段得到的用例来设计测试用例，以便测试导航设计的每个使用场景。根据用例和 -些 NSU(导航语义单元)，测试在界面外观中实现的导航机制(如菜单栏)，确保妨碍用例完成的错误被识别和纠正。

组件测试是测试 Web 应用中的内容单元和功能单元。注意 Web 应用中单元(Unit)的概念的转变。在内容体系结构中选择的单元就是网页,每个网页都封装了内容、导航链接和处理单元(表单、脚本、Applet 等)。Web 应用体系结构中的一个单元可能是一个被定义的功能组件,这个组件直接为最终用户提供服务;或可能是一个基础组件,这个组件使 Web 应用能完成它所有的能力。每个功能组件都使用与一般软件测试中测试单个模块一样的方法进行测试,在很多情况下都是黑盒测试。如果过程很复杂,也会使用白盒测试。除了功能测试,还要测试数据库的能力。

当构建 Web 应用体系结构时,导航和组件测试被用于集成测试。集成测试的策略依赖于被选择的内容和 Web 应用体系结构。如果内容体系结构被设计为线性、网格状或者简单的层次结构,就可以使用与通常软件的模块集成一样的方法集成 Web 页面。然而,如果我们使用混合的层次结构或者网状的体系结构,就可以采用在 OO(面向对象)测试中使用的集成测试方法。基于 Thread 的测试可以被用于集成 Web 页面集(一个 NSU 可以用来定义这样适合的集合)以便响应用户事件,每个 Thread 被单独集成和测试。回归测试用于确保修改没有带来副作用。Cluster 测试集成一组相互协作的页面(通过检测用例和 NSU 确定),生成测试用例用于发现页面协作中的错误。

Web 体系结构的每个组成部分要尽可能进行单元测试。比如,在 MVC 体系结构中 Model、View、Controller 组件,每一个都要被单独测试。集成之后,跨越这 3 部分的数据流和控制流都要经过仔细检查。

配置测试是要发现与某种特定的客户端和服务器环境相关的错误。构造一个交叉引用矩阵,定义所有可能的操作系统、浏览器、硬件平台、通信协议。要进行测试以发现与每一个可能的配置有关的错误。

安全测试包括一系列测试用来寻找 Web 应用和环境中易受攻击的漏洞,目的是要展示安全缺口是可能出现的。

性能测试包含设计一系列测试来评估:

(1) 增加用户量会怎样影响 Web 应用的响应时间和可靠性。

(2) 哪一个 Web 应用组件对性能下降负有责任和什么样的使用特征引起了性能退化。

(3) 性能退化如何影响 Web 应用的整体目标和需求。

10.3 内容测试

Web 应用内容的错误可能是微不足道的,如排版错误;也可能是严重的错误,如不正确的信息、不合适的组织结构、违反知识产权法等。内容测试是在用户遇到这些和其他很多的问题之前,首先发现它们。

内容测试结合评审和生成的可执行的测试用例两种方法。评审是发现内容中的语义错

误(10.3.1节讨论)。可执行的测试用例用于发现从一个或多个数据库中获取的数据驱动生成的动态内容中的错误。

10.3.1 内容测试目标

内容测试有3个重要目标:

(1) 发现文本文件、图像展示和其他媒体的句法错误(如排版错误,语法错误等)。

(2) 发现当进行导航时呈现在任何内容对象中的语义错误(即信息准确性和完整性的错误)。

(3) 找出呈现给最终用户的内容组织或结构上的错误。

要完成第一个目标,可以使用自动拼写和语法检查工具。不过,有许多句法错误无法被这些工具检查出来,因而需要专门评审人员(测试人员)才能查出来。根据前面所述,审稿是发现句法错误的最好方法。

语义测试关注于每个内容对象所呈现的信息。审查者或测试者必须回答以下这些问题:

- 这些信息准确无误吗?
- 这些信息是否简明扼要?
- 内容对象版面设计是否利于最终用户的理解?
- 内容对象中的信息是否容易被找到?
- 对于来自其他地方的所有信息是否已经提供恰当的引用说明?
- 展现的信息是否保持内部一致性,并且是否和其他的内容对象所展现的信息一致?
- 内容是否具有攻击性、容易引起误解或者容易引起法律诉讼?
- 内容是否侵犯了其他版权或商标?
- 内容中是否包含内部链接? 这些链接是否提供了相应的内容? 这些链接正确吗?
- 内容的审美风格是否与界面的审美风格冲突?

对一个大的Web应用(包括成百上千的内容对象)来说,回答所有这些问题是件很困难的事。但是,如果没有发现语义错误会动摇用户对Web应用的信心,因而会使Web应用失败。

存在于体系结构中的内容对象具有特殊的形式。在内容测试中,对内容组织和结构的测试,是为了确保所需的内容以恰当的顺序和关联关系呈现给最终用户。例如,cfi.ss.pku.edu.cn,该Web应用为项目经理、质量保证(QA)人员以及社区开发人员提供一个平台。项目经理可以在平台上制定项目计划、发布软件需求或软件设计、并招募软件开发人员;QA人员制定质量保证计划,审查、测试开发人员所提交的代码;社区开发人员应聘软件项目,依据项目计划和质量保证计划开发软件产品。CFI Web应用有大量关于软件项目的信息。内容对象提供描述性信息、技术规范说明、图片陈述和相关信息。对cfi.ss.pku.edu.cn的

内容架构进行测试是为了发现存在于这些信息中的错误(比如,对QA人员的问候语呈现给社区开发人员)。

10.3.2 数据库测试

现代Web应用不仅仅是显示静态的内容对象。在很多应用领域中,Web应用和复杂的数据管理系统交互并且实时地从数据库中获取数据从而动态地创建内容对象。

例如,一个金融服务Web应用能提供关于某项基于文本的、表格形式的和图形化的复杂的某特定资产(例如股票、共有基金)信息。当用户请求关于该资产的信息时,动态产生并提供这些信息内容对象。要达到上述目的需要做以下几步:

(1) 查询大型资产数据库。

(2) 从数据库中抽取相关数据。

(3) 抽取的数据必须被组织成内容对象。

(4) 这些内容对象(表达用户请求特定的信息)被发送到客户环境显示。

每一步都可能会发生错误,数据库测试的目标就是找到这些错误。

Web应用的数据库测试由于以下因素变得很复杂:

(1) 客户端对于信息的原始请求很少能够表现成直接输入数据库管理系统的格式(例如SQL)。所以要设计测试,来查找把用户请求转换成能够被数据库处理的格式过程中产生的错误。

(2) 数据库可以被远程服务器上的Web应用调用。所以,要测试Web应用和远端数据库通信中的错误。

(3) 从数据库得到的原始数据必须传送到服务器应用程序并且转换成恰当的格式以便将来传送给用户。因此,要设计测试用例来说明Web应用服务器接收到的原始数据的有效性,还要生成额外的测试用例来验证对原始数据进行有效的转换并创建有效的内容对象。

(4) 动态的内容对象必须能以某种能展示的形式传送给最终用户。所以,需要设计一系列的测试来找出内容对象格式中的错误和测试不同客户端环境配置下的兼容性。

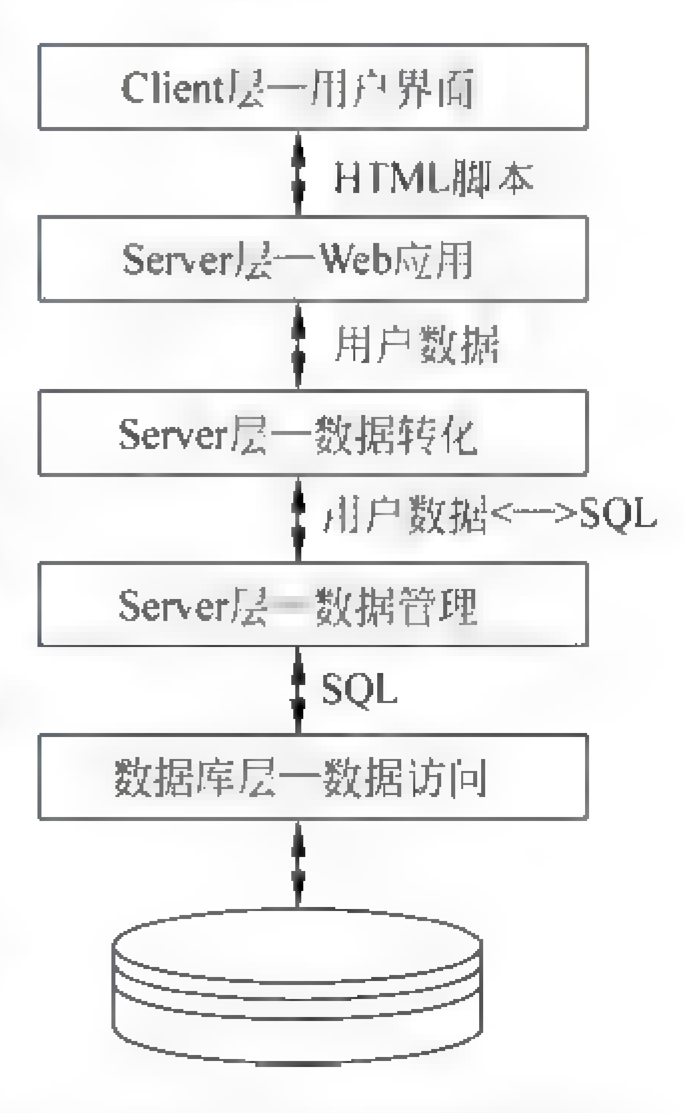


图 10-2 Web应用的交互层

考虑到上述4个因素,测试用例设计方法应该被应用到每一个"交互层",如图10-2所示。测试应该确保:

(1) 服务器和客户端之间通过界面层传输的信息有效。

(2) Web应用正确地处理脚本并恰当的抽取或者格式化用户数据。

(3) 用户数据可以正确地传送到服务器端,服务器端的

数据转换功能能够生成正确的查询(如 SQL)。

(4) 此查询被传输到负责与数据库存取例程(一般在另一台机器上)通信的数据管理层。

如图 10-2 所示的数据转化、数据管理、数据存取等层经常被实现为可重用的组件,并已经被独自确认以及作为一个整体确认过。这种情况下,Web 应用测试集中于设计测试用例来测试用户层和图中前两个服务器层(即 Web 应用层和数据转化层)的交互。

测试用户界面层是为了确保为每个用户查询被构造成恰当的 HTML 脚本,并将这些脚本妥当地传输到服务器端。测试服务器端的 Web 应用层来确保用户数据被恰当地从 HTML 脚本中抽离并且恰当地传输到服务器端的数据转化层。

测试数据转化功能是确保产生正确的 SQL,并且将之发送给相应的数据管理组件。这些技术与设计适当的数据库测试用例有关。这方面的详尽讨论已超出本书范围,感兴趣的读者可以参阅参考文献[5][6][7]。

> "作为电子消费者(E-Customers)(无论是商务的或是消费的),我们不可能对频繁遭受停机、事务处理中被挂起或者易用性感觉很差的网站有信心,因此测试在整个开发过程中扮演至关重要的角色。"
>
> Wing Lam

10.4 用户界面测试

在 Web 工程中,有 3 个不同的地方需要对 Web 应用进行用户界面的确认和验证。

(1) 在需求建模和分析阶段:评审界面模型以确保它们符合用户需求并符合分析模型的其他因素。

(2) 在设计阶段:评审界面设计模型以确保达到适用所有用户的一般性质量标准,并且妥当地解决了与应用有关的特殊界面的设计问题。

(3) 在测试阶段:重点转移到和应用相关的特殊的用户交互方面的执行,如界面的语法和语义所显示的那样。另外,测试也对易用性做最后评价。

10.4.1 界面测试策略

界面测试的总体策略是:找出与特定界面机制相关的错误(如,菜单链接不能正确执行的错误或数据输入表格方式的错误等)和找出在界面实现导航语义、Web 应用功能或内容显示方法中存在的错误。为了完成这个策略,必须实现下面的一系列目标:

(1) 测试界面特征(Feature)以确保设计规则、美观以及视觉内容对用户来说是可用

的，不存在错误，特征包括字体形状、颜色、结构、形象、边界、表格以及 Web 应用执行中所生成的相关元素。

(2) 类似于单元测试方式，对单个界面机制进行测试。比如，设计测试来检查所有的表单（Forms）、客户端脚本、动态 HTML、CGI 脚本、流内容和应用相关的界面机制（例如用于电子商务应用的购物车）。在很多情况下，测试可以完全集中在一个界面单元上，而不考虑其他界面的特性和功能。

(3) 针对一个特殊用户类别，使用某个用例或 NSU 对每个界面机制都进行测试。这种测试方法类似于集成测试，因为若干界面机制集合在一起以便执行那个用例或 NSU。

(4) 选定若干用例和 NSU，测试完整的界面以找出其中的语义错误。这种测试方法类似于确认测试，因为它的目的就是证明符合特定的用例或 NSU 语义。在这个阶段，要进行一系列易用性测试。

(5) 界面在各种环境中（比如浏览器）进行测试，以确保是兼容的。实际上，这一系列测试也可以看作是配置测试的一部分。

10.4.2 测试界面机制

当用户与 Web 应用交互时，交互是通过一个或多个界面机制进行的。下面对每个接口机制测试所需要考虑的事项进行简单描述[4]。

1. 链接

测试每个导航链接确保能到达恰当的内容对象或者实现相应的功能。Web 工程师建立与界面布局元素（比如菜单栏、索引项）有关的所有链接，然后分别执行；并且必须检测每一个内容对象中的链接以发现坏的 URL 或者连接到不恰当的内容对象和功能；最后对连接到外部 Web 应用的链接应该测试其精确性并评估链接超时情况下的风险。

2. 表单

从宏观上看，进行表单测试是为了确保：

(1) 标签正确表示表单内的域，必需的（Required）域对用户是可见的。

(2) 服务器收到表单内包含的所有信息，并且在客户端和服务器端传输时没有丢失数据。

(3) 当用户没有在下拉式菜单中选择或没有单击按钮时，系统会使用默认的选项。

(4) 浏览器功能（比如“后退”）不会破坏表单中输入的数据。

(5) 检查输入数据错误的脚本工作正常，并且提供有意义的错误信息。

在一个更高的级别上，表单测试要确保：

(1) 表单域有合适的宽度和数据类型。

(2) 当用户输入的字符串长度大于预先定义的最大长度时，表单已经建立了排除这种输入的保护措施。

(3) 下拉式菜单中选项的定义和排列要对最终用户来说要有意义。

(4) 浏览器自动填写的特性不要导致数据输入错误。

(5) Tab 键(或者其他的键)要在表单域之前进行合适的移动。

3. 客户端脚本

执行黑盒测试是为了发现脚本(如 Javascript)执行时发生的错误。这些测试附带着表单测试，因为脚本的输入经常来自于部分表单处理时提供的数据。兼容性测试应该被执行，从而确保已经被选择的脚本语言会在支持 Web 应用的环境配置下正常工作。除了测试脚本本身之外，Splaine 和 Jaskiel[4] 建议"你应该确保使用在公司标准里所选的并作为客户端(和服务器端)的脚本语言和版本"。

4. 动态 HTML

执行每个包含动态 HTML 的 Web 页面以确保动态显示是正确的，并且应该进行兼容性测试从而确保动态 HTML 在支持 Web 应用的环境配置下正确工作。

5. 弹出窗口

一系列的测试以确保：

(1) 弹出窗口大小合适，位置恰当。

(2) 弹出窗口不要覆盖原来的窗口。

(3) 弹出窗口的美观设计要与界面的美观设计一致。

(4) 附加在弹出窗口的滚动条和其他控制机制要位于合适的位置，并达到要求的功能。

6. CGI 脚本

执行黑盒测试的重点放在数据的完整性(当数据传输给 CGI 脚本的时候)和一旦接收到合法数据时的脚本处理。执行性能测试以确保服务器端配置能容纳处理多个 CGI 脚本的处理请求。

7. 流内容

测试应该证明流数据是最新的，能正确显示，能被毫无错误地挂起和毫无困难地重新开始。

8. Cookie

服务器端和客户端都需要进行测试。在服务器端，测试应该确保当特定的内容和功能

被请求时，正确地构建一个 Cookie(包含正确的数据)并正确传送给客户端。并且 Cookie 的持久性要经过测试，从而确保它的过期时间是正确的。在客户端，测试决定 Web 应用是否正确地把已经存在的 Cookie 附加到一个特定请求上(发送给服务器)。

9. 应用相关的界面机制

测试遵照一个由界面机制定义的功能和特性的检查表。比如，Splaine 和 Jaskiel[4] 建议对于一个为电子商务定义的购物车功能的检查表如下：

- 边界测试放到购物车中的最大和最小的商品数目。
- 测试对空购物车的结账请求。
- 测试从购物车中正确地删除一个商品。
- 测试确定是否购买之后购物车被清空了。
- 测试确定购物车内容的持久性(这应该作为客户需求被说明)。
- 测试确定如果客户请求被保存，Web 应用是否能在将来的某个时间"记起"购物车内的内容。

10.4.3 测试界面语义

一旦每个界面机制都被单元测试过，界面测试的重点就转到对界面语义的考虑。界面语义测试"评价设计是否很好地考虑了使用者，是否提供了清楚的方向、传递反馈，并保持语言和方法的一致性"[6]。

对界面设计模型的全面评审，可以对前面陈述中存在的隐含问题作出部分回答。然而，一旦 Web 应用被实现，每个用例场景(对于每一类用户)都必须被测试。本质上，一个用例成为一个测试序列设计的输入。测试序列的目的是发现错误，这些错误会使用户不能实现与用例相关的目标。

在每个用例被测试后，Web 项目组保留一个检查表以保证每个菜单项至少被测试一次，嵌入每一个内容对象的链接都被测试过。另外，测试顺序应该包括不正确的菜单选择与链接使用。其目的是要确定 Web 应用提供有效的错误处理与恢复功能。

10.4.4 易用性测试

易用性测试也评价用户和 Web 应用有效交互的程度和 Web 应用引导用户行为、提供有意义的反馈、强化界面访问一致性的程度。从这种意义上说，易用性测试类似于界面语义测试。然而易用性测试目的不在交互对象的语义上，易用性评审与测试是要确定 Web 应用界面方便用户使用的程度。

易用性测试可能由一个 Web 工程小组设计，但是测试用例本身由最终用户运行。有以

下步骤[4]：

(1) 定义一组易用性测试类型及其目标。

(2) 为评估每一个目标设计测试。

(3) 选择将要进行测试的参与者。

(4) 当执行测试时为参与者和 Web 应用之间的交互提供设备。

(5) 开发一种评价 Web 应用易用性的机制。

易用性测试可以用于多种不同抽象层次：

(1) 评价特定的界面机制(如表单)的易用性。

(2) 评价完整的网页(包括界面机制、数据对象和相关功能)的易用性。

(3) 评价整个 Web 应用的易用性。

易用性测试的第一步是识别易用性类别的集合，并为每一类别制定测试目标。以下的测试类别和对应目标举例说明了这一过程：

- 交互性——交互机制(例如，下拉式菜单、按钮、光标)是否易于理解和使用？
- 布局——布局风格是否使用户快速找到导航机制、内容、功能。
- 可读性——文档是否书写正确，便于理解？图形表述是否易于理解？
- 美观——布局、颜色、字体和相关特性是否易于使用？用户是否对 Web 应用感觉舒服？
- 展示特性——Web 应用是否使用了最优的屏幕尺寸和分辨率？
- 时间敏感性——重要的特性、功能和内容是否可以及时地使用和获得？
- 个性——Web 应用是否调整自身以适应不同类别、不同用户的需要？
- 无障碍性——有身体障碍的人是否可以理解 Web 应用？

对应以上每一种类型，设计了一系列测试。在一些情况下，这里的“测试”可能是视觉评审网页。在其他情况下，界面语义测试可能要再次执行，但是在这种情况下易用性关注的方面是极为重要的。

以对交互和界面机制的易用性评价为例。Constantine 和 Lockwood[8] 建议易用性测试应该检查下列界面元素：动画、按钮、颜色、控制、对话框、域、表单、窗框、图形、标签、链接、菜单、消息、导航条、页面、选择框、文本、工具栏。当评估每个特征时，执行测试的用户是按定性等级来进行评分。图 10-3 描述可由用户选择的评估“级别”集合。这些级别分别应用于各个特征、整个网页或整个 Web 应用。

10.4.5 兼容性测试

Web 应用必须运行于不同的环境。不同的计算机、显示设备、操作系统、浏览器和网络连接速度对 Web 应用的运行有重要的影响。每一种配置都会导致客户端运行速度、显示分辨率和网络连接速度的不同。操作系统反复无常的行为可能导致 Web 应用出现问题。不

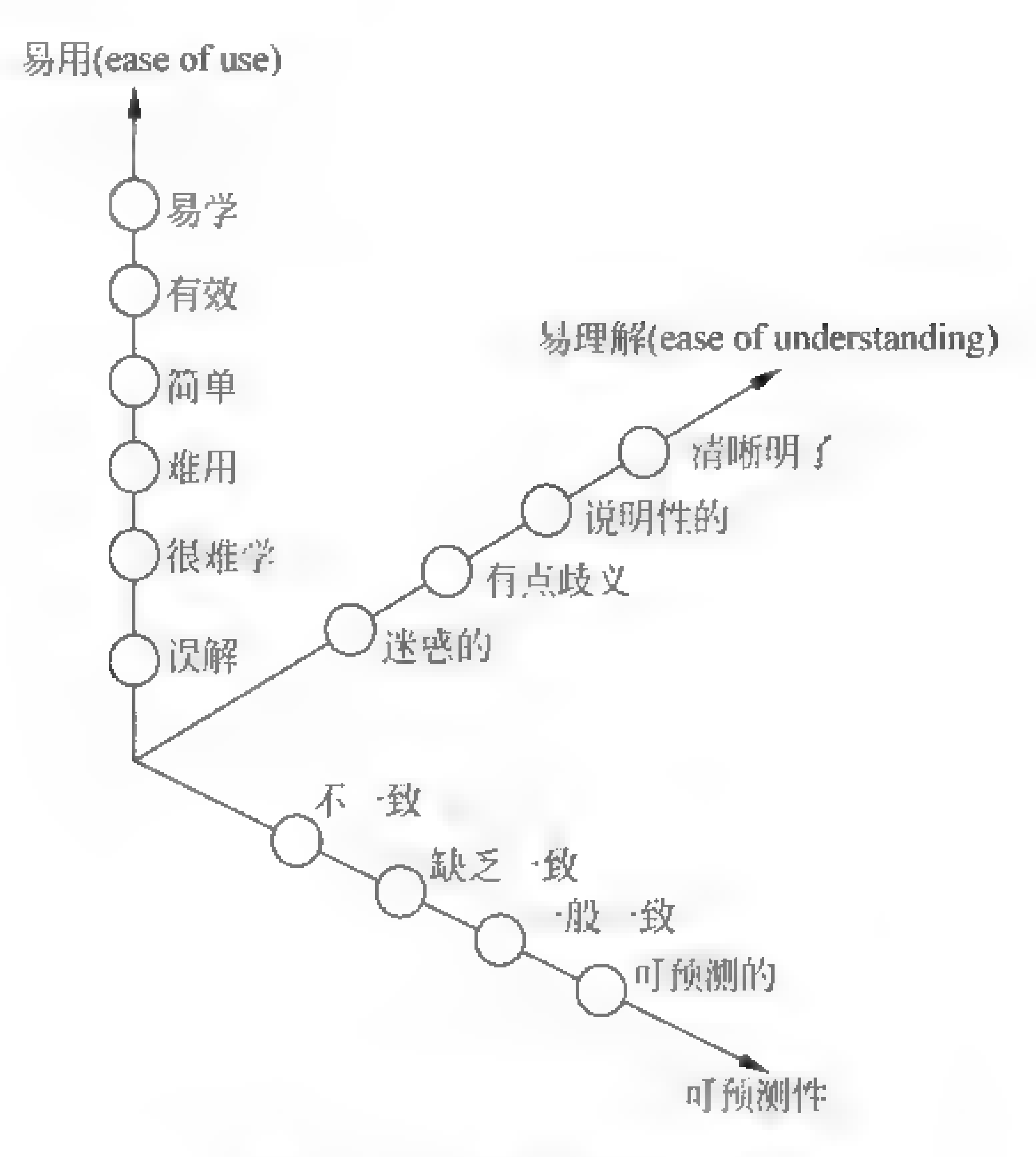


图 10-3 易用性的质量评价

考虑 Web 应用中 HTML 的标准化程度，不同的浏览器有时也会造成微妙的差异。对于特定的配置一些必需的插件可能或不能随即获得。

在有些情况下，细小的兼容性问题不会产生严重的问题，但是在另外一些情况下，这将产生严重的错误。例如，下载速度会变得慢得难以接受，缺少必要的插件使内容无法获得，浏览器的不同会造成引人注目的页面布局改变，字体风格可能被改变以至于难辨认的，或者表单可能被不合理地组织。兼容性测试努力在 Web 应用上线之前揭示这些问题。

兼容性测试的第一步是定义一个"经常遇到的"客户端计算机配置及其变化的集合，其本质是创立一个树结构。该结构定义每种计算平台、典型显示设备、平台支持的操作系统、可以获得的浏览器、可能的网络连接速度和类似的信息。下一步，Web 开发小组导出一系列兼容性确认测试，包括从现有的界面测试、导航测试、性能测试和安全测试。这些测试的目的是发现由配置不同引发的错误或运行时问题。

> Web 应用测试
>
> 地点：Henry Smith 的办公室。
>
> 人员：Henry Smith(CFI 软件开发小组经理)和 Alice Raman(CFI 软件开发小组组员)。
>
> 谈话记录：
>
> Henry：你认为 CFI Webapp V0.0 怎么样？
>
> Alice：外包方做得不错。Mary(vendor 开发负责人)告诉我他们正在按我们所说的方式进行测试。

Henry：我希望你和其他小组成员对这个电子商业站点做一点非正式的测试。
Alice(作苦相)：我还以为我们会雇佣第三方测试公司来确认这个Web应用。我们正在死撑着努力推出那个产品。
Henry：我们打算雇佣一个测试公司进行性能和安全测试，我们的外包测试公司已经开始测试了。我只是想换一种其他方法是否有帮助，还有，我们要控制成本，因此……
Alice(叹气)：你期盼着什么？
Henry：我想确定界面和所有的导航是可靠的。
Alice：我估计我们可以从每个主要界面的测试用例开始。

学习CFI系统
选择适合的CFI开发项目
应聘一个CFI开发项目
下载CFI项目开发设计

Henry：很好。但是要走完导航路径的全程一直到得到结果。
Alice：(浏览用例的记录本)是的，当选择"**选择适合的CFI开发项目**"时，将引导到达：

显示所有正在招聘的CFI开发项目
显示招聘单位信誉排名
选择一个CFI开发项目
我们可以测试每条路径的语义。

Henry：测试时，检查每一个导航节点的内容。
Alice：当然……还有功能单元。谁测试易用性？
Henry：哦……测试公司会进行易用性测试。我们已经雇了一家市场调查公司，他们列出了20种典型用户供易用性研究。不过如果你们发现任何易用性问题，也可以报告。
Alice：我知道，也报告给他们。
Henry：谢了，Alice。

10.5 组件级测试

组件级测试也叫功能测试，用来发现Web应用功能方面错误。每个Web应用功能都是一个软件模块(用一种程序或脚本语言实现)，可以使用黑盒测试技术(有时候用白盒技术)进行测试。

组件级别的测试用例经常由表单输入驱动。一旦定义了表单数据，用户通过选择一个按钮或其他界面控制机制开始执行测试。下面的测试用例设计方法是非常典型的：

• 等价类划分——功能的输入域被分成输入类别，由这些类别生成测试用例。评估输

入形式从而确定哪些数据类别和功能有关。当其他的输入类别不变时，为每一输入类别生成测试用例并执行测试用例。比如，一个电子商务应用可能实现一个计算运输费用的功能，通过表单提供多种关于用户邮政编码的运输信息。指定不同邮编号码会产生不同的错误，设计发现在处理邮编时产生错误的测试用例（比如一个不完整的邮编号码、一个正确的邮编号码、一个不存在的邮编号码、一个格式错误的邮编号码）。

- 边界值分析——测试表单数据的边界值。比如，前面提到的运输费用计算功能，需要产品运输的最长天数，表单里面的最短天数是 2 天、最长天数是 14 天。然而，边界值可能输入 0、1、2、13、14、15，从而确定功能对在有效输入的边界之外和之内的数据如何响应。
- 路径测试——如果功能的逻辑复杂度很高，那么使用路径测试（一种白盒测试用例设计方法）可确保程序中每个独立的路径都被测试到。

除了这些测试用例设计方法，使用强迫错误测试（Forced Error Testing）方法[6]生成测试用例，有目的地驱使 Web 组件进入一个错误的状态。这样的目的是发现错误处理时的错误（比如不正确的或不存在出错信息、由于错误导致 Web 应用失败、错误输入导致错误输出、与组件处理有关的副作用等）。

每个组件级别的测试用例指定组件提供的所有输入值和期望的输出。记录下测试所产生的实际结果输出，以便将来在技术支持和软件维护时参考。

在很多情况下，Web 应用功能的正确执行是与连接外部数据库的交互紧密联系的，因此，数据库测试成为组件级测试不可缺少的组成部分。Hower[9]这样写道：

> 数据库驱动的网站涉及在 Web 浏览器、操作系统、插件、通信协议、Web Server、数据库、（脚本语言）程序、安全增强（软件）、防火墙之间的复杂的交互。这种复杂性使得不可能测试网站的每一种依赖、每一个错误。典型的网站开发项目建立在一个具有“攻战式的”（Aggressive）进度表上，因此最好的测试方法会使用风险分析去确定哪里才是最需要测试的地方。风险分析应该包括考虑测试环境和真实环境的接近程度……风险分析的其他典型考虑如下：
>
> - 网站的哪个功能对其目的来说是最关键的？
> - 网站的哪一部分与数据库交互最多？
> - 网站的 CGI、Applets、ActiveX 等组件的哪些方面最复杂？
> - 什么问题会引起最多的抱怨或形成最糟的公众形象？
> - 网站的哪些方面是最流行的？
> - 网站的哪些方面有最高的安全风险？

当为 Web 应用组件和相关的数据库功能设计测试用例时，Hower 谈到的每一个风险都应该被考虑到。

10.6 导航测试

用户使用 Web 应用程序的过程就像旅游者游览博物馆，将有多条路线可供选择，用户可能多次停下来，有许多事情需要学习与研究，开始一些活动，然后做一些决定。像前面讨论的，如果旅游者是带着一些目的来的，那么导航过程在一定意义上是可预测的。同时，导航过程又可能是不可预测的，这是因为旅游者会受他所看到听到的事物的影响，可能在开始某些活动时改变了初衷。所以导航测试的目的是：

(1) 保证任何用户可以使用的路径都处于可工作状态。

(2) 确认每个导航语义单元都可被适当类型的用户使用。

> *“我们没有迷路，只是被所处的位置影响了。”*
>
> John M. Ford

10.6.1 测试导航语法

导航测试的第一个阶段实际上始于界面测试。对各种导航机制进行测试以保证每一个都完成它们本身的功能。Splaine 和 Jaskie[4] 建议下面的每一种导航机制都要被测试到：

- 导航链接——包括 Web 应用程序中的内部链接，指向其他应用程序的外部链接和特定网页中的锚点。通过测试保证选择链接时可以获取相应的内容以及实现相应的功能。
- 重定向——是当用户请求不存在的 URL 或选择的链接目标被删除了或名字被改变的情况下发生的。向用户展示一条消息，导航重新被指向另一个页面(例如主页)。重定向应当通过请求不正确的内部链接或外部 URL 来进行测试，还要对程序的相应处理进行评测。
- 书签——虽然书签是属于浏览器的功能，但也应当测试 Web 应用程序保证创建书签时可以提取到有意义的网页标题。
- 框架——每个框架都包含特定网页的内容；一个框架集包含多个框架，并可以同时展现多个网页。因为一个框架或框架集可能存在于另一个之中，所以对这些导航和展现机制应当进行测试，看是否可以获得正确的内容、合适的外观与大小、下载的性能以及浏览器的兼容性。
- 网站地图——应当测试入口以保证通过链接使用户得到正确的内容和合适的功能。
- 内部搜索引擎——复杂的 Web 应用程序经常包含成百上千的内容对象。一个内部

搜索引擎允许用户通过关键字搜索得到需要的内容。内部搜索引擎测试是测试搜索的精确性和完整性，搜索引擎的错误处理以及高级搜索特性（比如在搜索域中使用布尔操作符）。

上述的一些测试可以通过自动化工具完成（例如链接检验），而其他一些则需要人工设计和执行。导航测试的最终目的是保证在 Web 应用投入使用前发现导航机制中的各种错误。

10.6.2 测试导航语义

前面也提到了导航语义单元（NSU），这里给出它的定义"导航语义单元是信息及其相关导航结构的集合，它们为了完成相关的用户需求而相互协作"[10]。每个 NSU 被一组导航路径（称作"导航路"）所定义，导航路径链接导航节点（比如 Web 页面、内容对象、功能）。从整体看，每个 NSU 允许用户完成特定的需求，这种需求由某类用户的一个或多个用例定义。导航测试对每个 NSU 进行测试，确保这些需求被满足。

当每个 NSU 被测试时，Web 项目组必须回答下面的问题：

- NSU 是不是无误地实现了它的完整性？
- 在为 NSU 定义的导航路径上下文中每个导航节点是不是可达的？
- 如果 NSU 可以被多个导航路径达到，那么是不是每个相关路径都被测试到了？
- 如果用户界面为辅助导航提供了指导帮助，那么在导航时这些指导是否正确而可理解的吗？
- 除了浏览器的 back 按钮之外，有返回上一层节点和路径首节点的机制吗？
- 一个大的导航节点（如很长的网页）内部的导航机制工作正常吗？
- 如果在一个节点的某个功能被执行，而用户没有提供输入，那么余下的 NSU 会被完成吗？
- 如果在某个节点执行功能时发生了错误，那么这个 NSU 能否被完成？
- 有没有方法可以在所有节点被到达之前暂停导航？并且能不能返回暂停的节点继续导航？
- 网站地图的节点是否都可以达到？最终用户明白这些节点名称的意思吗？
- 如果从某个外部的节点到达一个 NSU 内部的节点，能在导航路径上到达下一个节点吗？能返回到先前的导航路径上的节点吗？
- 当执行 NSU 时，用户能理解当前所在内容结构的位置吗？

导航测试与界面测试和可用行测试一样，应该有尽可能多的人参与。早期由 Web 开发工程师进行，后期由其他涉众、独立的测试小组进行，最后由非技术类的用户进行。目的就是全面进行 Web 应用的导航测试。

10.7 配置测试

Web应用具有很大的挑战性，其中配置的变化和不稳定性是很重要的因素。对一个用户来说，硬件、操作系统、浏览器、存储能力、网络通信速度以及其他的客户端因素等都是很难预测的。并且，对一个给定的用户来讲，配置(操作系统升级、新的因特网提供商及连接速度)以一定的规则发生变化，导致易出错的客户端环境，错误的影响可能是微妙而重要的。如果两个用户的客户端环境配置不同，那么一个用户对Web应用的印象和跟Web应用的交互方式与另一个用户会有很大不同。

配置测试的工作不是要把每一种可能的客户端配置都测试到，而是测试一组最可能的客户端和服务器端配置以保证在所有的配置上用户体验都相同，并把和某一种配置相关的错误隔离出来。

10.7.1 服务器端问题

在服务器端，设计配置测试用例是为了验证服务器端的计划配置(比如Web服务器、数据库服务器、操作系统、防火墙、并发应用等)能否正确无误地支持Web应用。本质上，Web应用安装在服务器端，测试是为了在服务器端发现配置相关的错误。

当设计服务器端配置测试时，Web工程师应该考虑服务器配置的每个组件。下面是在服务器端配置测试中需要提出并要回答的问题：

- Web应用完全与服务器操作系统兼容吗？
- 当Web应用运行时，系统文件、目录、相关的系统数据会被正确创建吗？
- 系统安全措施允许Web应用执行，对用户的服务没有受到干扰或造成性能下降吗？
- 当选择分布式服务器配置后Web应用被测试了吗？
- Web应用能与数据库软件集成吗？Web应用对不同版本的数据库软件敏感吗？
- Web应用脚本会被正确执行吗？
- 系统管理员的错误对Web应用的影响被测试了吗？
- 如果使用代理服务器，那么是否考虑了在线测试时它们的不同配置会带来什么影响？

10.7.2 客户端问题

在客户端，配置测试集中在Web应用配置的兼容性上。配置包括下列组件的一个或多个置换排列组合[6]：

- 硬件：CPU、内存、存储器、打印设备。
- 操作系统：Linux 操作系统、Macintosh 操作系统、Microsoft Windows 操作系统、基于移动设备的操作系统。
- 浏览器软件：Internet Explorer、Mozilla/Netscape、Opera、Safari 以及其他浏览器。
- 用户界面组件：ActiveX、Java applets、SVG 以及其他界面组件。
- 插件：QuickTime、RealPlayer、SVG Viewer 以及很多其他插件。
- 网络连接设备及技术：有线、DSL、调制解调器、T1。

除了这些组件以外，其他变量还包括网络软件、各种 ISP 的差异和应用程序并发运行等。

为了设计客户端配置测试，Web 工程小组必须减少配置变量数目到一个可管理的程度。为了完成这些测试，评估每一类用户从而确定这类用户可能遇到的配置。并且可以使用行业共享的数据预测最可能的组件组合，从而在这些环境中进行配置测试。

10.8 安全测试

Web 应用的安全是一个复杂的主题，要想完成有效的安全测试，必须对这一主题有全面深入的理解。Web 应用以及服务器端和客户端的运行环境吸引了很多人的注意，如外部的黑客、爱抱怨的雇员、不诚实的竞争者或任何人，他希望盗取敏感信息，恶意更改内容，降低(系统)性能，使功能不起作用，或使一个人、组织、业务染上麻烦。

> "在因特网上进行商业活动、存储财产是有风险的，黑客、解密高手、网络偷窃者、网络哄骗者、小偷、故意破坏者、病毒传播者、流氓软件可能会泛滥。"
>
> Dorothy and Peter Denning

设计安全测试是为了探查在客户端的漏洞和网络在客户端与服务器端传输数据时漏洞以及服务器端的漏洞。这些域的每一部分都可能被攻击，安全测试人员就是为了发现那些可能会被某些带有企图的人利用的漏洞。

客户端的漏洞经常能追溯到存在于浏览器、邮件程序、通信软件中的缺陷。Nguyen[6] 描述了一个典型的安全漏洞：

一个经常提到的缺陷是缓冲区溢出。内存溢出允许客户端机器执行恶意代码。比如，在浏览器里输入 URL 时，当长度超过分配给 URL 的缓冲区大小时，如果浏览器没有错误检测代码来确认输入 URL 长度，将会导致内存覆盖(缓冲区溢出)错误。一个老练的黑客常利用这个缺陷通过写一个长 URL 并带有可执行代码，从而引起浏览器崩溃或更改安全设置(从高级别到低级别)，更糟糕的是，还可能会破坏用户的数据。

另外一个客户端的漏洞是未经授权去访问浏览器中的cookie。带恶意目的的网站可以获取包含在合法的cookies中的信息，利用这些信息危害用户的隐私，更糟糕的是，达到盗取用户身份的目的。

客户和服务器之间通信的数据很容易被骗取。当一端的通信路径被一个怀有恶意的实体破坏就会发生骗取，比如，一个用户可以会被一个恶意的网站骗取，这个恶意网站的外观可能与合法的Web应用服务器一样，目的是盗取客户的密码、私有信息、信用卡数据。

服务器端的漏洞包括拒绝服务攻击和恶意脚本。恶意脚本可以传给客户或使服务器操作不起作用，并且服务器数据库可能会被未经授权访问（数据盗取）。

为了防止这些（或更多的）漏洞，应实现下面的一个或更多安全要素。

- 防火墙：是一种软硬件结合的过滤机制，检测每一个输入包确保它来源的合法性，阻止可疑的包。
- 认证：是一种验证机制，确认客户和服务器的真实身份，只有两边都被确认之后才能允许通信。
- 加密：是一种编码机制，保护敏感数据不被怀有恶意目的地读写和更改。数字认证增强了加密的功能，允许客户验证数据被传输到的目的地。
- 授权：是一种过滤机制，只有那些具有合法授权代码（比如用户ID和密码）的用户才能访问客户端和服务器端。

安全测试的目的是暴露那些可能被怀有恶意的人利用的安全要素中的漏洞。安全测试设计要求对每个安全要素的内在工作原理有很深的理解并对网络技术有很广泛深入的理解。在很多情况下，安全测试一般都外包给专门的公司。

10.9 性能测试

在竞争者的网站只需数秒的事情，而自己的Web应用下载内容需要几分钟，没有比这样的事情更让人感到沮丧的了；在登录一个网站时收到"服务器正忙"，并建议用户稍后再试，没有比这样的事情更让人烦恼的了；在一些情况下，Web应用立即响应，而一些情况下就好像进入无限等待状态，没有比这样的事情更让人不安的了。所有这些事情每天都会发生在Web上，且都与性能有关。

性能测试是要发现性能问题。性能问题可以来自服务器端缺乏资源、不足的网络带宽、数据库的性能不足、操作系统不完善、糟糕的Web应用功能设计和其他软、硬件问题，这些都会导致客户端-服务器性能的下降。测试有两方面的目的：

（1）理解系统怎样响应负载（也就是说用户数、事务数或数据量）。

（2）收集为了提高性能而更改设计的指标。

10.9.1 性能测试目标

性能测试是为了模拟现实世界的负载状况。当同时请求的用户数增加，或者在线事务数增加，或者下载或上传的数据量增加时，性能测试能帮助回答下面的问题：

- 服务器的响应时间会下降到值得注意和不可接受的程度吗？
- 在哪一点(根据用户数、事务数或数据负载量)性能变得不可接受了？
- 系统的哪些组件导致性能下降？
- 在负载变化时用户的平均响应时间是多少？
- 性能下降对系统安全有影响吗？
- 随着系统负载增长时 Web 应用的可靠性和准确性会受到影响吗？
- 当负载大于系统的最大容量时会发生什么情况？

为了回答上面这些问题，采用两种性能测试方法。

- 负载测试：根据负载级别和组合的变化，对现实的负载进行测试。
- 压力测试：负载增加到转折点时确定 Web 应用环境能处理多大的容量。

下面分别考虑这两个测试策略。

10.9.2 负载测试

负载测试的目的是确定 Web 应用和它的服务器端环境如何响应各种各样的负载条件。进行测试时，下面变量的排列组合定义了一个测试条件集：

- N——并发的用户数。
- T——单位时间每个用户的在线事务数。
- D——每个事务被服务器处理的数据负载量。

每种情况下，这些变量定义在系统正常的操作范围内。当运行每种测试条件时，收集一个或者更多的测试量度：平均的用户响应、下载标准单元数据的平均时间、处理一个事务的平均时间。Web 项目组检查这些量度从而确定性能的急速下降可以追溯到的一个特定的 N、T、D 组合。

负载测试可以用于评估给 Web 应用用户推荐的连接速度。用下面的方法计算吞吐量 P 值。

$$P = N \times T \times D$$

考虑一个流行的体育新闻网站，在一个给定的时刻，平均每 2 分钟 20000 个并发用户同时提交一个请求(事务 T)，每个事务要求 Web 应用下载一篇平均大小 3KB 的新文章。因此，这样计算吞吐量：

$$P = (20000 \times 0.5 \times 3\text{KB})/60 = 500\text{Kbytes/秒} = 4\text{Mbits/秒}(\text{注：1bytes} = 8\text{bits})$$

对于服务器的网络连接来说,必须提供数据传输率的支持,并且要确保通过测试。

10.9.3 压力测试

压力测试是负载测试的继续,但是在这里 N、T、D 变量要满足和超过操作限制。这些测试的目的是要回答下面的问题:

- 当超出系统容量时系统是逐渐退化还是服务器关闭吗?
- 服务软件会出现"服务器不可用"的信息吗?更一般地说,用户会意识到服务器不能访问吗?
- 服务器对请求进行排队吗?当服务请求取消后服务器会清空请求队列吗?
- 容量超出限制时事务会丢失吗?
- 容量超出限制时数据完整性会受到影响吗?
- 多大的 N、T、D 值会引起服务器环境失败?怎么声明这些失败?警告信息会自动发给服务器网站的技术支持人员吗?
- 如果系统宕机了,多长时间才能恢复?
- 当服务器容量达到 80%~90%时,Web 应用程序的某些功能(比如计算加强功能、数据流能力)会不会不能继续运行?

压力测试的一个变种有时叫做脉冲反弹(spike/bounce)测试[4]。这种测试首先把负载设为系统容量,然后快速降到正常水平,接着再回到系统容量。通过反弹设定系统负载,测试人员能确定服务器是否能够调度资源满足负载高峰的需求,然后恢复到正常负载时又能够释放资源(为了准备下一个高峰)。

10.10 总结

Web 应用测试是测试 Web 应用质量的每个纬度,目的是发现错误或者发现导致质量失败的问题。测试集中在内容、功能、结构、易用性、导航、性能、兼容性、互操作、容量、安全等方面。测试应该和 Web 应用设计的评审相结合。

Web 应用测试策略从内容、功能、导航等的单元测试开始,测试质量的每个纬度。一旦每个单元被确认了,重点转移到测试 Web 应用的整体;为了达到这个目标,很多测试从用户的角度出发,由用例中包含的信息驱动。开发一个 Web 工程测试计划包括识别测试步骤、工件(比如测试用例)和评估测试结果的机制。测试过程包括 7 个不同的测试类型。

内容测试(和评审)集中在各种各样的内容类型,目的是发现影响内容准确性语义和句法的错误或者呈现给最终用户的方式错误。界面测试是测试用户和 Web 应用的交互机制和确认界面的美观,目的是发现在界面语义中由于交互机制的不良实现、省略、不一致、二义

性而引起的错误。

导航测试应用在软件分析阶段得到的用例。设计测试用例要根据导航设计执行每个使用场景。测试导航机制确保妨碍用例完成的任何错误被确认和更正。组件测试是测试Web应用的内容和功能单元。每个网页封装内容、导航链接、处理元素，它们形成Web应用体系结构中构成“单元”，这些单元必须被测试。

配置测试是要发现与某一特殊的客户端或服务器端环境相关的错误或者兼容性问题，进行测试从而发现与每个可能配置有关的错误。安全测试组织一系列测试以利用Web应用和其环境的弱点，目的是发现安全漏洞。性能测试包含一系列评估Web应用中当服务器资源的需求增加时，评估响应时间和可靠性的测试。

10.11 参考文献

[1] Wallace, D. ,I. Raggett,J. Aufgang. *Extreme Programming for Web Projects*. Addition-Wesley. 2003

[2] Miler, E.. WebSite Testing. 2000. http://www. soft. com/evalid/Technology/White. Papers/website. testing. html

[3] Nguyen, H.. Testing Web-based Applications. Software Testing and Quality Engineering,May/June, 2000,http://www. stqemagazine. com

[4] Splaine, S. ,S. Jaskiel. *The Wb Testing Handbook*. STQE Publishing. 2001

[5] Sceppa, D.. *Microsoft ADO. NET*. Microsoft Press,2001

[6] Nguyen, H.. *Testing Applications on the Web*. Wiley. 2001

[7] Brown, B.. *Oracle9i Web Development*. McGraw-Hill. 2nd. 2001

[8] Constantine, L. , L. Lockwood. *Softwar for Use*. Addison-Wesley, 1999. see also http://www. foruse. com/

[9] Hower, Rick, Beyond Broken Links, Internet System, 1997. http://www. dbmsmag. com/ 9707I03. html

[10] Cacheri, C. ,et al.. Conceptual Navigation Analysis: A Device and Platform Independent Navigation Specification. Proc. 2nd Intl. Workshop on Web-Oriented Technology, June 2002. download from http://www. dsic. upv. es/～west/iwwost02/papers/cachero. pdf

10.12 思考与练习

1. 试用你自己的话，描述Web工程项目中，测试的目标是什么?

2. 讨论哪种错误更为严重，是客户端错误还是服务器端错误? 为什么?

3. 对Web应用的什么元素可以进行“单元”测试? 哪类测试必须在集成Web应用的元素之后进行?

4. 以下的描述是否是绝对的："测试 Web 应用的总体策略是从用户可视的元素开始，然后向技术元素推进"？有没有一些例外？

5. 内容测试是通常意义上的"真正"测试吗？请给出解释。

6. 描述 Web 应用数据库测试的步骤。在客户端还是服务器端数据库测试是一主导活动？

7. 界面机制测试与界面语义测试的区别是什么？

8. 导航语法与导航语义的区别是什么？

9. 安全测试的目的是什么？由谁担任这项测试？

10. 负载测试与压力测试的区别是什么？

10.13 进一步阅读

Lydia Ash, The Web Testing Companion: The Insider's Guide to Efficient and Effective Tests. 1st edition. Wiley, 2003

Elfriede Dustin, Jeff Rashka, Douglas McDiarmid. Quality Web Systems: Performance, Security, and Usability. Addison-Wesley Professional, 2001

Hung Q. Nguyen. Testing Applications on the Web. 2nd Edition. Wiley, 2003

Steven Splaine, Stefan P. Jaskiel. The Web Testing Handbook. S T Q E Pub, 2001

Steven Splaine. Testing Web Security: Assessing the Security of Web Sites and Applications. 1 edition Wiley, 2002

读者意见反馈

亲爱的读者：

感谢您一直以来对清华版计算机教材的支持和爱护。为了今后为您提供更优秀的教材，请您抽出宝贵的时间来填写下面的意见反馈表，以便我们更好地对本教材做进一步改进。同时如果您在使用本教材的过程中遇到了什么问题，或者有什么好的建议，也请您来信告诉我们。

地址：北京市海淀区双清路学研大厦 A 座 602 室　计算机与信息分社营销室　收

邮编：100084　　电子邮件：jsjjc@tup. tsinghua. edu. cn

电话：010-62770175-4608/4409　　邮购电话：010-62786544

教材名称：软件测试方法与实践

ISBN 978-7-302-18458-4

个人资料

姓名：__________ 年龄：______所在院校/专业：__________

文化程度：________ 通信地址：__________

联系电话：________ 电子信箱：__________

您使用本书是作为：□指定教材 □选用教材 □辅导教材 □自学教材

您对本书封面设计的满意度：

□很满意 □满意 □一般 □不满意　改进建议__________

您对本书印刷质量的满意度：

□很满意 □满意 □一般 □不满意　改进建议__________

您对本书的总体满意度：

从语言质量角度看　□很满意 □满意 □一般 □不满意

从科技含量角度看　□很满意 □满意 □一般 □不满意

本书最令您满意的是：

□指导明确 □内容充实 □讲解详尽 □实例丰富

您认为本书在哪些地方应进行修改？(可附页)

____

____

您希望本书在哪些方面进行改进？(可附页)

____

____

电子教案支持

敬爱的教师：

为了配合本课程的教学需要，本教材配有配套的电子教案(素材)，有需求的教师可以与我们联系，我们将向使用本教材进行教学的教师免费赠送电子教案(素材)，希望有助于教学活动的开展。相关信息请拨打电话 010-62776969 或发送电子邮件至 jsjjc@tup. tsinghua. edu. cn 咨询，也可以到清华大学出版社主页(http://www. tup. com. cn 或 http://www. tup. tsinghua. edu. cn)上查询。